IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION

FOR

UNITED STATES PATENT

FOR

PROGRAMMABLE PROCESSOR AND SYSTEM FOR PARTITIONED FLOATING-POINT MULTIPLY-ADD OPERATION

INVENTORS:

Craig Hansen
Los Altos, California

John Moussouris
Palo Alto, California

SPECIFICATION

RELATED APPLICATIONS

[0001] This application is a continuation of U.S. Patent Application No. 10/646,787, filed August 25, 2003, which is a continuation of U.S. Patent Application No. 09/922,319, filed August 2, 2001, which is a continuation of U.S. Patent Application No. 09/382,402, filed August 24, 1999, now U.S. Patent No. 6,295,599, which claims the benefit of priority to Provisional Application No 60/097,635 filed August 24, 1998, and is a continuation-in-part of U.S. Patent Application No. 09/169,963, filed October 13, 1998, now U.S. Patent No. 6,006,318, which is a continuation of U.S. Patent Application No. 08/754,827, filed November 22, 1996 now U.S. Patent No. 5,822,603, which is a divisional of U.S. Patent Application No. 08/516,036, filed August 16, 1995 now U.S. Patent No. 5,742,840.

REFERENCE TO PARENT APPLICATIONS

[0002] The contents of U. S. Patent Application Nos. 09/382,402 and 09/922,319 are hereby incorporated by reference including their appendices in their entirety.

REFERENCE TO AN APPENDIX

[0003] This application includes an appendix, submitted herewith .The contents of the appendix are hereby incorporated by reference.

FIELD OF THE INVENTION

[0004] The present invention relates to general purpose processor architectures, and particularly relates to general purpose processor architectures capable of executing group operations.

BACKGROUND OF THE INVENTION

[0005] The performance level of a processor, and particularly a general purpose processor, can be estimated from the multiple of a plurality of interdependent factors: clock rate, gates per clock, number of operands, operand and data path width, and

operand and data path partitioning. Clock rate is largely influenced by the choice of circuit and logic technology, but is also influenced by the number of gates per clock. Gates per clock is how many gates in a pipeline may change state in a single clock cycle. This can be reduced by inserting latches into the data path: when the number of gates between latches is reduced, a higher clock is possible. However, the additional latches produce a longer pipeline length, and thus come at a cost of increased instruction latency. The number of operands is straightforward; for example, by adding with carry-save techniques, three values may be added together with little more delay than is required for adding two values. Operand and data path width defines how much data can be processed at once; wider data paths can perform more complex functions, but generally this comes at a higher implementation cost. Operand and data path partitioning refers to the efficient use of the data path as width is increased, with the objective of maintaining substantially peak usage.

Summary of the invention

[0006] Embodiments of the invention pertain to systems and methods for enhancing the utilization of a general purpose processor by adding classes of instructions. These classes of instructions use the contents of general purpose registers as data path sources, partition the operands into symbols of a specified size, perform operations in parallel, catenate the results and place the catenated results into a general-purpose register. Some embodiments of the invention relate to a general purpose microprocessor which has been optimized for processing and transmitting media data streams through significant parallelism.

[0007] Some embodiments of the present invention provide a system and method for improving the performance of general purpose processors by including the capability to execute group operations involving multiple floating-point operands. In one embodiment, a programmable media processor comprises a virtual memory addressing unit, a data path, a register file comprising a plurality of registers coupled to the data path, and an execution unit coupled to the data path capable of executing group-floating point operations in which multiple floating-point operations stored in partitioned fields of

one or more of the plurality of registers are operated on to produce catenated results. The group floating-point operations may involve operating on at least two of the multiple floating-point operands in parallel. The catenated results may be returned to a register, and general purpose registers may used as operand and result registers for the floating-point operations. In some embodiments the execution unit may also be capable of performing group floating-point operations on floating-point data of more than one precision. In some embodiments the group floating-point operations may include group add, group subtract, group compare, group multiply and group divide arithmetic operations that operate on catenated floating-point data. In some embodiments, the group floating-point operations may include group multiply-add, group scale-add, and group set operations that operate on catenated floating-point data.

[0008] In one embodiment, the execution unit is also capable of executing group integer instructions involving multiple integer operands stored in partitioned fields of registers. The group integer operations may involve operating on at least two of the multiple integer operands in parallel. The group integer operations may include group add, group subtract, group compare, and group multiply arithmetic operations that operate on catenated integer data.

[0009] In one embodiment, the execution unit is capable of performing group data handling operations, including operations that copy, operations that shift, operations that rearrange and operations that resize catenated integer data stored in a register and return catenated results. The execution unit may also be configurable to perform group data handling operations on integer data having a symbol width of 8 bits, group data handling operations on integer data having a symbol width of 16 bits, and group data handling operations on integer data having a symbol width of 32 bits. In one embodiment, the operations are controlled by values in a register operand. In one embodiment, the operations are controlled by values in the instruction.

[0010] In one embodiment, the multi-precision execution unit is capable of executing a Galois field instruction operation.

[0011] In one embodiment, the multi-precision execution unit is configurable to execute a plurality of instruction streams in parallel from a plurality of threads, and the programmable media processor further comprises a register file associated with each thread executing in parallel on the multi-precision execution unit to support processing of the plurality of threads. In some embodiments, the multi-precision execution unit executes instructions from the plurality of instruction streams in a round-robin manner. In some embodiments, the processor ensures only one thread from the plurality of threads can handle an exception at any given time.

[0012] Some embodiments of the present invention provide a multiplier array that is fully used for high precision arithmetic, but is only partly used for other, lower precision operations. This can be accomplished by extracting the high-order portion of the multiplier product or sum of products, adjusted by a dynamic shift amount from a general register or an adjustment specified as part of the instruction, and rounded by a control value from a register or instruction portion. The rounding may be any of several types, including round-to-nearest/even; toward zero, floor, or ceiling. Overflows are typically handled by limiting the result to the largest and smallest values that can be accurately represented in the output result.

[0013] When an extract is controlled by a register, the size of the result can be specified, allowing rounding and limiting to a smaller number of bits than can fit in the result. This permits the result to be scaled for use in subsequent operations without concern of overflow or rounding. As a result, performance is enhanced. In those instances where the extract is controlled by a register, a single register value defines the size of the operands, the shift amount and size of the result, and the rounding control. By placing such control information in a single register, the size of the instruction is reduced over the number of bits that such an instruction would otherwise require, again improving performance and enhancing processor flexibility. Exemplary instructions are Ensemble Convolve Extract, Ensemble Multiply Extract, Ensemble Multiply Add Extract, and Ensemble Scale Add Extract. With particular regard to the Ensemble Scale Add Extract Instruction, the extract control information is combined in a register with two values used

as scalar multipliers to the contents of two vector multiplicands. This combination reduces the number of registers otherwise required, thus reducing the number of bits required for the instruction.

[0014] In one embodiment, the processor performs load and store instructions operable to move values between registers and memory. In one embodiment, the processor performs both instructions that verify alignment of memory operands and instructions that permit memory operands to be unaligned. In one embodiment, the processor performs store multiplex instructions operable to move to memory a portion of data contents controlled by a corresponding mask contents. In one embodiment, this masked storage operation is performed by indivisibly reading-modifying-writing a memory operand.

[0015] In one embodiment, all processor, memory and interface resources are directly accessible to high-level language programs. In one embodiment, assembler codes and high-level language formats are specified to access enhanced instructions. In one embodiment interface and system state is memory mapped, so that it can be manipulated by compiled code. In one embodiment, software libraries provide other operations required by the ANSI/IEEE floating-point standard. In one embodiment, software conventions are employed at software module boundaries, in order to permit the combination of separately compiled code and to provide standard interfaces between application, library and system software. In one embodiment, instruction scheduling is performed by a compiler.

THE FIGURES

[0016] Figure 1 is a system level diagram showing the functional blocks of a system according to the present invention.

[0017] Figure 2 is a matrix representation of a wide matrix multiply in accordance with one embodiment of the present invention.

[0018] Figure 3 is a further representation of a wide matrix multiple in accordance with one embodiment of the present invention.

[0019] Figure 4 is a system level diagram showing the functional blocks of a system incorporating a combined Simultaneous Multi Threading and Decoupled Access from Execution processor in accordance with one embodiment of the present invention.

[0020] Figure 5 illustrates a wide operand in accordance with one embodiment of the present invention.

[0021] Figure 6 illustrates an approach to specifier decoding in accordance with one embodiment of the present invention.

[0022] Figure 7 illustrates in operational block form a Wide Function Unit in accordance with one embodiment of the present invention.

[0023] Figure 8 illustrates in flow diagram form the Wide Microcache control function.

[0024] Figure 9 illustrates Wide Microcache data structures.

[0025] Figures 10 and 11 illustrate a Wide Microcache control.

[0026] Figure 12 is a timing diagram of a decoupled pipeline structure in accordance with one embodiment of the present invention.

[0027] Figure 13 further illustrates the pipeline organization of Figure 12.

[0028] Figure 14 is a diagram illustrating the basic organization of the memory management system according to the present embodiment of the invention.

[0029] Figure 15 illustrates the physical address of an LTB entry for thread th, entry en, byte b.

[0030] Figure 16 illustrates a definition for AccessPhysicalLTB.

[0031] Figure 17 illustrates how various 16-bit values are packed together into a 64-bit LTB entry.

[0032] Figure 18 illustrates global access as fields of a control register.

[0033] Figure 19 shows how a single-set LTB context may be further simplified by reserving the implementation of the lm and la registers.

[0034] Figure 20 shows the partitioning of the virtual address space if the largest possible space is reserved for an address space identifier.

[0035] Figure 21 shows how the LTB protect field controls the minimum privilege level required for each memory action of read (r), write (w), execute (x), and gateway (g), as well as memory and cache attributes of write allocate (wa), detail access (da), strong ordering (so), cache disable (cd), and write through (wt).

[0036] Figure 22 illustrates a definition for LocalTranslation.

[0037] Figure 23 shows how the low-order GT bits of the th value are ignored, reflecting that 2GT threads share a single GTB.

[0038] Figure 24 illustrates a definition for AccessPhysicalGTB.

[0039] Figure 25 illustrates the format of a GTB entry.

[0040] Figure 26 illustrates a definition for GlobalAddressTranslation.

[0041] Figure 27 illustrates a definition for GTBUpdateWrite.

[0042] Figure 28 shows how the low-order GT bits of the th value are ignored, reflecting that 2GT threads share single GTB registers.

[0043] Figure 29 illustrates the registers GTBLast, GTBFirst, and GTBBump.

[0044] Figure 30 illustrates a definition for AccessPhysicalGTBRegisters.

[0045] Figures 31A-31C illustrate Group Boolean instructions in accordance with an exemplary embodiment of the present invention.

[0046] Figures 31D-31E illustrate Group Multiplex instructions in accordance with an exemplary embodiment of the present invention.

[0047] Figures 32A - 32C illustrate Group Add instructions in accordance with an exemplary embodiment of the present invention.

[0048] Figures 33A - 33C illustrate Group Subtract and Group Set instructions in accordance with an exemplary embodiment of the present invention.

[0049] Figures 34A - 34C illustrate Ensemble Divide and Ensemble Multiply instructions in accordance with an exemplary embodiment of the present invention.

[0050] Figures 35A - 35C illustrate Group Compare instructions in accordance with an exemplary embodiment of the present invention.

[0051] Figures 36A - 36C illustrate Ensemble Unary instructions in accordance with an exemplary embodiment of the present invention.

[0052] Figure 37 illustrates exemplary functions that are defined for use within the detailed instruction definitions in other sections.

[0053] Figures 38A - 38C illustrate Ensemble Floating-Point Add, Ensemble Floating-Point Divide, and Ensemble Floating-Point Multiply instructions in accordance with an exemplary embodiment of the present invention.

[0054] Figures 38D – 38F illustrate Ensemble Floating-Point Multiply Add instructions in accordance with an exemplary embodiment of the present invention.

[0055] Figures 38G – 38I illustrate Ensemble Floating-Point Scale Add instructions in accordance with an exemplary embodiment of the present invention.

[0056] Figures 39A - 39C illustrate Ensemble Floating-Point Subtract instructions in accordance with an exemplary embodiment of the present invention.

[0057] Figures 39D – 39G illustrate Group Set Floating-point instructions in accordance with an exemplary embodiment of the present invention.

[0058] Figures 40A - 40C illustrate Group Compare Floating-point instructions in accordance with an exemplary embodiment of the present invention.

[0059] Figures 41A - 41C illustrate Ensemble Unary Floating-point instructions in accordance with an exemplary embodiment of the present invention.

[0060] Figures 42A - 42D illustrate Ensemble Multiply Galois Field instructions in accordance with an exemplary embodiment of the present invention.

[0061] Figures 43A - 43D illustrate Compress, Expand, Rotate, and Shift instructions in accordance with an exemplary embodiment of the present invention.

[0062] Figures 43E – 43G illustrate Shift Merge instructions in accordance with an exemplary embodiment of the present invention.

[0063] Figures 43H – 43J illustrate Compress Immediate, Expand Immediate, Rotate Immediate, and Shift Immediate instructions in accordance with an exemplary embodiment of the present invention.

[0064] Figures 43K – 43M illustrate Shift Merge Immediate instructions in accordance with an exemplary embodiment of the present invention.

[0065] Figures 44A - 44D illustrate Extract instructions in accordance with an exemplary embodiment of the present invention.

[0066] Figures 44E – 44G illustrate Ensemble Extract instructions in accordance with an exemplary embodiment of the present invention.

[0067] Figures 45A - 45F illustrate Deposit and Withdraw instructions in accordance with an exemplary embodiment of the present invention.

[0068] Figures 45G – 45J illustrate Deposit Merge instructions in accordance with an exemplary embodiment of the present invention.

[0069] Figures 46A - 46E illustrate Shuffle instructions in accordance with an exemplary embodiment of the present invention.

[0070] Figures 47A - 47C illustrate Swizzle instructions in accordance with an exemplary embodiment of the present invention.

[0071] Figures 47D – 47E illustrate Select instructions in accordance with an exemplary embodiment of the present invention.

[0072] Figure 48 is a pin summary describing the functions of various pins in accordance with the one embodiment of the present invention.

[0073] Figures 49A-49G present electrical specifications describing AC and DC parameters in accordance with one embodiment of the present invention.

[0074] Figures 50A - 50C illustrate Load instructions in accordance with an exemplary embodiment of the present invention.

[0075] Figures 51A - 51C illustrate Load Immediate instructions in accordance with an exemplary embodiment of the present invention.

[0076] Figures 52A - 52C illustrate Store and Store Multiplex instructions in accordance with an exemplary embodiment of the present invention.

[0077] Figures 53A - 53C illustrate Store Immediate and Store Multiplex Immediate instructions in accordance with an exemplary embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[0078] Referring first to Figure 1, a general purpose processor is illustrated therein in block diagram form. In Figure 1, four copies of an access unit are shown, each with an access instruction fetch queue A-Queue 101-104. Each access instruction fetch queue A-Queue 101104 is coupled to an access register file AR 105-108, which are each coupled to two access functional units A 109-116. In a typical embodiment, each thread of the processor may have on the order of sixty-four general purpose registers (e.g., the AR's 105-108 and ER's 125-128). The access units function independently for four simultaneous threads of execution, and each compute program control flow by performing arithmetic and branch instructions and access memory by performing load and store instructions. These access units also provide wide operand specifiers for wide operand instructions. These eight access functional units A 109-116 produce results for

access register files AR 105-108 and memory addresses to a shared memory system 117-120.

[0079] In one embodiment, the memory hierarchy includes on-chip instruction and data memories, instruction and data caches, a virtual memory facility, and interfaces to external devices. In Figure 1, the memory system is comprised of a combined cache and niche memory 117, an external bus interface 118, and, externally to the device, a secondary cache 119 and main memory system with I/O devices 120. The memory contents fetched from memory system 117-120 are combined with execute instructions not performed by the access unit, and entered into the four execute instruction queues E-Queue 121-124. In accordance with one embodiment of the present invention, from the software perspective, the machine state includes a linear byte-addressed shared memory space. For wide instructions, memory contents fetched from memory system 117-120 are also provided to wide operand microcaches 132-136 by bus 137. Instructions and memory data from E-queue 121-124 are presented to execution register files 125-128, which fetch execution register file source operands. The instructions are coupled to the execution unit arbitration unit Arbitration 131, that selects which instructions from the four threads are to be routed to the available execution functional units E 141 and 149, X 142 and 148, G 143-144 and 146-147, and T 145. The execution functional units E 141 and 149, the execution functional units X 142 and 148, and the execution functional unit T 145 each contain a wide operand microcache 132-136, which are each coupled to the memory system 117 by bus 137.

[0080] The execution functional units G 143-144 and 146-147 are group arithmetic and logical units that perform simple arithmetic and logical instructions, including group operations wherein the source and result operands represent a group of values of a specified symbol size, which are partitioned and operated on separately, with results catenated together. In a presently preferred embodiment the data path is 128 bits wide, although the present invention is not intended to be limited to any specific size of data path.

[0081] The execution functional units X 142 and 148 are crossbar switch units that perform crossbar switch instructions. The crossbar switch units 142 and 148 perform data handling operations on the data stream provided over the data path source operand buses 151-158, including deal, shuffles, shifts, expands, compresses, swizzles, permutes and reverses, plus the wide operations discussed hereinafter. In a key element of a first aspect of the invention, at least one such operation will be expanded to a width greater than the general register and data path width. Examples of the data manipulation operations are described in the Appendix included herewith.

[0082] The execution functional units E 141 and 149 are ensemble units that perform ensemble instructions using a large array multiplier, including group or vector multiply and matrix multiply of operands partitioned from data path source operand buses 151-158 and treated as integer, floating-point, polynomial or Galois field values. According to the present embodiment of the invention, a general software solution is provided to the most common operations required for Galois Field arithmetic. The instructions provided include a polynomial multiply, with the polynomial specified as one register operand. This instruction can be used to perform CRC generation and checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding. Also, matrix multiply instructions and other operations described in the Appendix included herewith utilize a wide operand loaded into the wide operand microcache 132 and 136.

[0083] The execution functional unit T 145 is a translate unit that performs table-look-up operations on a group of operands partitioned from a register operand, and catenates the result. The Wide Translate instruction described in the Appendix included herewith utilizes a wide operand loaded into the wide operand microcache 134.

[0084] The execution functional units E 141, 149, execution functional units X -142, 148, and execution functional unit T each contain dedicated storage to permit storage of source operands including wide operands as discussed hereinafter. The dedicated storage 132-136, which may be thought of as a wide microcache, typically has

a width which is a multiple of the width of the data path operands related to the data path source operand buses 151-158. Thus, if the width of the data path 151-158 is 128 bits, the dedicated storage 132-136 may have a width of 256, 512, 1024 or 2048 bits. Operands which utilize the full width of the dedicated storage are referred to herein as wide operands, although it is not necessary in all instances that a wide operand use the entirety of the width of the dedicated storage; it is sufficient that the wide operand use a portion greater than the width of the memory data path of the output of the memory system 117-120 and the functional unit data path of the input of the execution functional units 141-149, though not necessarily greater than the width of the two combined. Because the width of the dedicated storage 132-136 is greater than the width of the memory operand bus 137, portions of wide operands are loaded sequentially into the dedicated storage 132-136. However, once loaded, the wide operands may then be used at substantially the same time. It can be seen that functional units 141-149 and associated execution registers 125-128 form a data functional unit, the exact elements of which may vary with implementation.

[0085] The execution register file ER 125-128 source operands are coupled to the execution units 141-145 using source operand buses 151-154 and to the execution units 145-149 using source operand buses 155-158. The function unit result operands from execution units 141145 are coupled to the execution register file ER 125-128 using result bus 161 and the function units result operands from execution units 145-149 are coupled to the execution register file using result bus 162.

[0086] The wide operands used in some embodiments of the present invention provide the ability to execute complex instructions such as the wide multiply matrix instruction shown in Figure 2, which can be appreciated in an alternative form, as well, from Figure 3. As can be appreciated from Figures 2 and 3, a wide operand permits, for example, the matrix multiplication of various sizes and shapes which exceed the data path width. The example of Figure 2 involves a matrix specified by register rc having a 128*64/size multiplied by a vector contained in register rb having a 128 size, to yield a result, placed in register rd, of 128 bits.

[0087] The operands that are substantially larger than the data path width of the processor are provided by using a general-purpose register to specify a memory specifier from which more than one but in some embodiments several data path widths of data can be read into the dedicated storage. The memory specifier typically includes the memory address together with the size and shape of the matrix of data being operated on. The memory specifier or wide operand specifier can be better appreciated from Figure 5, in which a specifier 500 is seen to be an address, plus a field representative of the size/2 and a further field representative of width/2, where size is the product of the depth and width of the data. The address is aligned to a specified size, for example sixty-four bytes, so that a plurality of low order bits (for example, six bits) are zero. The specifier 500 can thus be seen to comprise a first field 505 for the address, plus two field indicia 510 within the low order six bits to indicate size and width.

[0088] The decoding of the specifier 500 may be further appreciated from Figure 6 where, for a given specifier 600 made up of an address field 605 together with a field 610 comprising plurality of low order bits. By a series of arithmetic operations shown at steps 615 and 620, the portion of the field 610 representative of width/2 is developed. In a similar series of steps shown at 625 and 630, the value of t is decoded, which can then be used to decode both size and address. The portion of the field 610 representative of size/2 is decoded as shown at steps 635 and 640, while the address is decoded in a similar way at steps 645 and 650.

[0089] The wide function unit may be better appreciated from Figure 7, in which a register number 700 is provided to an operand checker 705. Wide operand, specifier 710 communicates with the operand checker 705 and also addresses memory 715 having a defined memory width. The memory address includes a plurality of register operands 720A-n, which are accumulated in a dedicated storage portion 714 of a data functional unit 725. In the exemplary embodiment shown in Figure 7, the dedicated storage 714 can be seen to have a width equal to eight data path widths, such that eight wide operand portions 730A-H are sequentially loaded into the dedicated storage to form the wide

operand. Although eight portions are shown in Figure 7, the present invention is not limited to eight or any other specific multiple of data path widths. Once the wide operand portions 730A-H are sequentially loaded, they may be used as a single wide operand 735 by the functional element 740, which may be any element(s) from Figure 1 connected thereto. The result of the wide operand is then provided. to a result register 745, which in a presently preferred embodiment is of the same width as the memory width.

[0090] Once the wide operand is successfully loaded into the dedicated storage 714, a second aspect of the present invention may be appreciated. Further execution of this instruction or other similar instructions that specify the same memory address can read the dedicated storage to obtain the operand value under specific conditions that determine whether the memory operand has been altered by intervening instructions. Assuming that these conditions are met, the memory operand fetch from the dedicated storage is combined with one or more register operands in the functional unit, producing a result. In some embodiments, the size of the result is limited to that of a general register, so that no similar dedicated storage is required for the result. However, in some different embodiments, the result may be a wide operand, to further enhance performance.

[0091] To permit the wide operand value to be addressed by subsequent instructions specifying the same memory address, various conditions must be checked and confirmed:

[0092] Those conditions include:

1. Each memory store instruction checks the memory address against the memory addresses recorded for the dedicated storage. Any match causes the storage to be marked invalid, since a memory store instruction directed to any of the memory addresses stored in dedicated storage 714 means that data has been overwritten.

2. The register number used to address the storage is recorded. If no intervening instructions have written to the register, and the same register is used on the subsequent instruction, the storage is valid (unless marked invalid by rule #1).

3. If the register has been modified or a different register number is used, the value of the register is read and compared against the address recorded for the dedicated storage. This uses more resources than #1 because of the need to fetch the register contents and because the width of the register is greater than that of the register number itself. If the address matches, the storage is valid. The new register number is recorded for the dedicated storage.

[0093] If conditions #2 or #3 are not met, the register contents are used to address the general-purpose processor's memory and load the dedicated storage. If dedicated storage is already fully loaded, a portion of the dedicated storage must be discarded (victimized) to make room for the new value. The instruction is then performed using the newly updated dedicated storage. The address and register number is recorded' for the dedicated storage.

[0094] By checking the above conditions, the need for saving and restoring the dedicated storage is eliminated. In addition, if the context of the processor is changed and the new context does not employ Wide instructions that reference the same dedicated storage, when the original context is restored, the contents of the dedicated storage are allowed to be used without refreshing the value from memory, using checking rule #3. Because the values in the dedicated storage are read from memory and not modified directly by performing wide operations, the values can be discarded at any time without saving the results into general memory. This property simplifies the implementation of rule #4 above.

[0095] An alternate embodiment of the present invention can replace rule #1 above with the following rule:

1a. Each memory store 'instruction checks the memory address against the memory addresses recorded for the dedicated storage. Any match causes the dedicated storage to be updated, as well as the general memory.

[0096] By use of the above rule 1.a, memory store instructions can modify the dedicated storage, updating just the piece of the dedicated storage that has been changed, leaving the remainder intact. By continuing to update the general memory, it is still true that the contents of the dedicated memory can be discarded at any time without saving the results into general memory. Thus rule #4 is not made more complicated by this choice. The advantage of this alternate embodiment is that the dedicated storage need not be discarded (invalidated) by memory store operations.

[0097] Referring next to Figure 9, an exemplary arrangement of the data structures of the wide microcache or dedicated storage 114 may be better appreciated. The wide microcache contents, wmc.c, can be seen to form a plurality of data path widths 900A-n, although in the example shown the number is eight. The physical address, wmc.pa, is shown as 64 bits in the example shown, although the invention is not limited to a specific width. The size of the contents, wmc.size, is also provided in a field which is shown as 10 bits in an exemplary embodiment. A "contents valid" flag, wmc.ev, of one bit is also included in the data structure, together with a two bit field for thread last used, or wmc.th. In addition, a six bit field for register last used, wmc.reg, is provided in an exemplary embodiment. Further, a one bit flag for register and thread valid, or wmc.rtv, may be provided.

[0098] The process by which the microcache is initially written with a wide operand, and thereafter verified as valid for fast subsequent operations, may be better appreciated from Figure 8. The process begins at 800, and progresses to step 805 where a check of the register contents is made against the stored value wmc.rc. If true, a check is made at step 810 to verify the thread. If true, the process then advances to step 815 to verify whether the register and thread are valid. If step 815 reports as true, a check is made at step 820 to verify whether the contents are valid. If all of steps 805 through 820

return as true, the subsequent instruction is able to utilize the existing wide operand as shown at step 825, after which the process ends. However, if any of steps 805 'through 820 return as false, the process branches to step 830, where content, physical address and size are set. Because steps 805 through 820 all lead to either step 825 or 830, steps 805 through 820 may be performed in any order or simultaneously without altering the process. The process then advances to step 835 where size is checked. This check basically ensures that the size of the translation unit is greater than or equal to the size of the wide operand, so that a physical address can directly replace the use of a virtual address. The concern is that, in some embodiments, the wide operands may be larger than the minimum region that the virtual memory system is capable of mapping. As a result, it would be possible for a single contiguous virtual address range to be mapped into multiple, disjoint physical address ranges, complicating the task of comparing physical addresses. By determining the size of the wide operand and comparing that size against the size of the virtual address mapping region which is referenced, the instruction is aborted with an exception trap if the wide operand is larger than the mapping region. This ensures secure operation of the processor. Software can then re-map the region using a larger size map to continue execution if desired. Thus, if size is reported as unacceptable at step 835, an exception is generated at step 840. If size is acceptable, the process advances to step 845 where physical address is checked. If the check reports as met, the process advances to step 850, where a check of the contents valid flag is made. If either check at step 845 or 850 reports as false, the process branches and new content is written into the dedicated storage 114, with the fields thereof being set accordingly. Whether the check at step 850 reported true, or whether new content was written at step 855, the process advances to step 860 where appropriate fields are set to indicate the validity of the data, after which the requested function can be performed at step 825. The process then ends.

[0099] Referring next to Figures 10 and 11, which together show the operation of the microcache controller from a hardware standpoint, the operation of the microcache controller may be better understood. In the hardware implementation, it is clear that conditions which are indicated as sequential steps in Figure 8 and 9 above can be

performed in parallel, reducing the delay for such wide operand checking. Further, a copy of the indicated hardware may be included for each wide microcache, and thereby all such microcaches as may be alternatively referenced by an instruction can be tested in parallel. It is believed that no further discussion of Figures 10 and 11 is required in view of the extensive discussion of Figures '8 and 9, above.

[00100] Various alternatives to the foregoing approach do exist for the use of wide operands, including an implementation in which a single instruction can accept two wide operands, partition the operands into symbols, multiply corresponding symbols together, and add the products to produce a single scalar value or a vector of partitioned values of width of the register file, possibly after extraction of a portion of the sums. Such an instruction can be valuable for detection of motion or estimation of motion in video compression. A further enhancement of such an instruction can incrementally update the dedicated storage if the address of one wide operand is within the range of previously specified wide operands in the dedicated storage, by loading only the portion not already within the range and shifting the in-range portion as required. Such an enhancement allows the operation to be performed over a "sliding window" of possible values. In such an instruction, one wide operand is aligned and supplies the size and shape information, while the second wide operand, updated incrementally, is not aligned.

[00101] Another alternative embodiment of the present invention can define additional instructions where the result operand is a wide operand. Such an enhancement removes the limit that a result can be no larger than the size of a general register, further enhancing performance. These wide results can be cached locally to the functional unit that created them, but must be copied to the general memory system before the storage can be reused and before the virtual memory system alters the mapping of the address of the wide result. Data paths must be added so that load operations and other wide operations can read these wide results - forwarding of a wide result from the output of a functional unit back to its input is relatively easy, but additional data paths may have to be introduced if it is desired to forward wide results back to other functional units as wide operands.

[00102] As previously discussed, a specification of the size and shape of the memory operand is included in the low-order bits of the address. In a presently preferred implementation, such memory operands are typically a power of two in size and aligned to that size. Generally, one-half the total size is added (or inclusively or'ed, or exclusively or'ed) to the memory address, and one half of the data width is added (or inclusively or'ed, or exclusively or'ed) to the memory address. These bits can be decoded and stripped from the memory address, so that the controller is made to step through all the required addresses. This decreases the number of distinct operands required for these instructions, as the size, shape and address of the memory operand are combined into a single register operand value.

[00103] Particular examples of wide operations which are defined by the present invention include the Wide Switch instruction that performs bit-level switching; the Wide Translate instruction which performs byte (or larger) table-lookup; Wide Multiply Matrix, Wide Multiply Matrix Extract and Wide Multiply Matrix Extract Immediate (discussed below), Wide Multiply Matrix Floating-point, and Wide Multiply Matrix Galois (also discussed below). While the discussion below focuses on particular sizes for the exemplary instructions, it will be appreciated that the invention is not limited to a particular width.

[00104] The Wide Switch instruction rearranges the contents of up to two registers (256 bits) at the bit level, producing a full-width (128 bits) register result. To control the rearrangement, a wide operand specified by a single register, consisting of eight bits per bit position is used. For each result bit position, eight wide operand bits for each bit position select which of the 256 possible source register bits to place in the result. When a wide operand size smaller than 128 bytes, the high order bits of the memory operand are replaced with values corresponding to the result bit position, so that the memory operand specifies a bit selection within symbols of the operand size, performing the same operation on each symbol.

[00105] The Wide Translate instructions use a wide operand to specify a table of depth up to 256 entries and width of up to 128 bits. The contents of a register is partitioned into operands of one, two, four, or eight bytes, and the partitions are used to select values from the table in parallel. The depth and width of the table can be selected by specifying the size and shape of the wide operand as described above.

[00106] The Wide Multiply Matrix instructions use a wide operand to specify a matrix of values of width up to 64 bits (one half of register file and data path width) and depth of up to 128 bits/symbol size. The contents of a general register (128 bits) is used as a source operand, partitioned into a vector of symbols, and multiplied with the matrix, producing a vector of width up to 128 bits of symbols of twice the size of the source operand symbols. The width and depth of the matrix can be selected by specifying the size and shape of the wide operand as described above. Controls within the instruction allow specification of signed, mixed-signed, unsigned, complex, or polynomial operands.

[00107] The Wide Multiply Matrix Extract instructions use a wide operand to specify a matrix of value of width up to 128 bits (full width of register file and data path) and depth of up to 128 bits/symbol size. The contents of a general register (128 bits) is used as a source operand, partitioned into a vector of symbols, and multiplied with the matrix, producing a vector of width up to 256 bits of symbols of twice the size of the source operand symbols plus additional bits to represent the sums of products without overflow. The results are then extracted in a manner described below (Enhanced Multiply Bandwidth by Result Extraction), as controlled by the contents of a general register specified by the instruction. The general register also specifies the format of the operands: signed, mixed-signed, unsigned, and complex as well as the size of the operands, byte (8 bit), doublet (16 bit), quadlet (32 bit), or hexlet (64 bit).

[00108] The Wide Multiply Matrix Extract Immediate instructions perform the same function as above, except that the extraction, operand format and size is controlled by fields in the instruction. This form encodes common forms of the above instruction without the need to initialize a register with the required control information. Controls

within the instruction allow specification of signed, mixed-signed, unsigned, and complex operands.

[00109] The Wide Multiply Matrix Floating-point instructions perform a matrix multiply in the same form as above, except that the multiplies and additions are performed in floating-point arithmetic. Sizes of half (16-bit), single (32-bit), double (64-bit), and complex sizes of half, single and double can be specified within the instruction.

[00110] Wide Multiply Matrix Galois instructions perform a matrix multiply in the same form as above, except that the multiples and additions are performed in Galois field arithmetic. A size of 8 bits can be specified within the instruction. The contents of a general register specify the polynomial with which to perform the Galois field remainder operation. The nature of the matrix multiplication is novel and described in detail below.

[00111] In another aspect of the invention, memory operands of either little-endian or big-endian conventional byte ordering are facilitated. Consequently, all Wide operand instructions are specified in two forms, one for little-endian byte ordering and one for big-endian byte ordering, as specified by a portion of the instruction. The byte order specifies to the memory system the order in which to deliver the bytes within units of the data path width (128 bits), as well as the order to place multiple memory words (128 bits) within *a* larger Wide operand. Each of these instructions is described in greater detail in the Appendix filed herewith.

[00112] Some embodiments of the present invention address extraction of a high order portion of a multiplier product or sum of products, as a way of efficiently utilizing a large multiplier array. Parent U.S. Patent No. 5,742,840 and U.S. Patent No. 5,953,241 describe a system and method for enhancing the utilization of a multiplier array by adding specific classes of instructions to a general-purpose processor. This addresses the problem of making the most use of a large multiplier array that is fully used for high-precision arithmetic - for example a 64x64 bit multiplier is fully used by a 64-bit by

64-bit multiply, but only one quarter used for a 32-bit by 32-bit multiply) for (relative to the multiplier data width and registers) low-precision arithmetic operations. In particular, operations that perform a great many low-precision multiplies which are combined (added) together in various ways are specified. One of the overriding considerations in selecting the set of operations is a limitation on the size of the result operand. In an exemplary embodiment, for example, this size might be limited to on the order of 128 bits, or a single register, although no specific size limitation need exist.

[00113] The size of a multiply result, a product, is generally the sum of the sizes of the operands, multiplicands and multiplier. Consequently, multiply instructions specify operations in which the size of the result is twice the size of identically-sized input operands. For our prior art design, for example, a multiply instruction accepted two 64-bit register sources and produces a single 128-bit register-pair result, using an entire 64x64 multiplier array for 64-bit symbols, or half the multiplier array for pairs of 32-bit symbols, or one-quarter the multiplier array for quads of 16-bit symbols. For all of these cases, note that two register sources of 64 bits are combined, yielding a 128-bit result.

[00114] In several of the operations, including complex multiplies, convolve, and matrix multiplication, low-precision multiplier products are added together. The additions further increase the required precision. The sum of two products requires one additional bit of precision; adding four products requires two, adding eight products requires three, adding sixteen products requires four. In some prior designs, some of this precision is lost, requiring scaling of the multiplier operands to avoid overflow, further reducing accuracy of the result.

[00115] The use of register pairs creates an undesirable complexity, in that both the register pair and individual register values must be bypassed to subsequent instructions. As a result, with prior art techniques only half of the source operand 128-bit register values could be employed toward producing a single-register 128-bit result.

[00116] In some embodiments of the present invention, a high-order portion of the multiplier product or sum of products is extracted, adjusted by a dynamic shift amount from a general register or an adjustment specified as part of the instruction, and, rounded by a control value from a register or instruction portion as round-to-nearest/even, toward zero, floor, or ceiling. Overflows are handled by limiting the result to the largest and smallest values that can be accurately represented in the output result. This operation is more fully described in the attached Appendix.

[00117] In the present invention, when the extract is controlled by a register, the size of the result can be specified, allowing rounding and limiting to a smaller number of bits than can fit in the result. This permits the result to be scaled to be used in subsequent operations without concern of overflow or rounding, enhancing performance.

[00118] Also in the present invention, when the extract is controlled by a register, a single register value defines the size of the operands, the shift amount and size of the result, and the rounding control. By placing all this control information in a single register, the size of the instruction is reduced over the number of bits that such a instruction would otherwise require, improving performance and enhancing flexibility of the processor.

[00119] The particular instructions included in this aspect of the present invention are Ensemble Convolve Extract, Ensemble Multiply Extract, Ensemble Multiply Add Extract and Ensemble Scale Add Extract, each of which is more thoroughly treated in the appendix.

[00120] An aspect of the present invention defines the Ensemble Scale Add Extract instruction, that combines the extract control information in a register along with two values that are used as scalar multipliers to the contents of two vector multiplicands. This combination reduces the number of registers that would otherwise be required, or the number of bits that the instruction would otherwise require, improving performance.

[00121] Several of these instructions (Ensemble Convolve Extract, Ensemble Multiply Add Extract) are typically available only in forms where the extract is specified as part of the instruction. An alternative embodiment can incorporate forms of the operations in which the size of the operand, the shift amount and the rounding can be controlled by the contents of a general register (as they are in the Ensemble Multiply Extract instruction). The definition of this kind of instruction for Ensemble Convolve Extract, and Ensemble Multiply Add Extract would require four source registers, which increases complexity by requiring additional general-register read ports.

[00122] Another alternative embodiment can reduce the number of register read-ports required for implementation of instructions in which the size, shift and rounding of operands is controlled by a register. The value of the extract control register can be fetched using an additional cycle on an initial execution and retained within or near the functional unit for subsequent executions, thus reducing the amount of hardware required for implementation with a small additional performance penalty. The value retained would be marked invalid, causing a re-fetch of the extract control register, by instructions that modify the register, or alternatively, the retained value can be updated by such an operation. A re-fetch of the extract control register would also be required if a different register number were specified on a subsequent execution. It should be clear that the properties of the above two alternative embodiments can be combined.

[00123] Another embodiment of the invention includes Galois field arithmetic, where multiplies are performed by an initial binary polynomial multiplication (unsigned binary multiplication with carries suppressed), followed by a polynomial modulo/remainder operation (unsigned binary division with carries suppressed). The remainder operation is relatively expensive in area and delay. In Galois field arithmetic, additions are performed by binary addition with carries suppressed, or equivalently, a bitwise exclusive-or operation. In this aspect of the present invention, a matrix multiplication is performed using Galois field arithmetic, where the multiplies and additions are Galois field multiples and additions.

[00124] Using prior art methods, a 16 byte vector multiplied by a 16x16 byte matrix can be performed as 256 8-bit Galois field multiplies and 16*15=240 8-bit Galois field additions. Included in the 256 Galois field multiplies are 256 polynomial multiplies and 256 polynomial remainder operations. But by use of the present invention, the total computation can be reduced significantly by performing 256 polynomial multiplies, 240 16-bit polynomial additions, and 16 polynomial remainder operations. Note that the cost of the polynomial additions has been doubled, as these are now 16-bit operations, but the cost of the polynomial remainder functions has been reduced by a factor of 16. Overall, this is a favorable tradeoff, as the cost of addition is much lower than the cost of remainder.

[00125] In a still further aspect of the present invention, a technique is provided for incorporating floating point information into processor instructions. In U.S. Patent No. 5,812,439, a system and method are described for incorporating control of rounding and exceptions for floating-point instructions into the instruction itself. The present invention extends this invention to include separate instructions in which rounding is specified, but default handling of exceptions is also specified, for a particular class of floating-point instructions. Specifically, the SINK instruction (which converts floating-point values to integral values) is available with control in the instruction that include all previously specified combinations (default-near rounding and default exceptions, Z - round-toward-zero and trap on exceptions, N - round to nearest and trap on exceptions, F - floor rounding (toward minus infinity) and trap on exceptions, C - ceiling rounding (toward plus infinity) and trap on exceptions, and X - trap on inexact and other exceptions), as well as three new combinations (Z.D - round toward zero and default exception handling, F.D - floor rounding and default exception handling, and C.D - ceiling rounding and default exception handling). (The other combinations: N.D is equivalent to the default, and X.D - trap on inexact but default handling for other exceptions is possible but not particularly valuable).

Pipelining and Multithreading

[00126] As shown in Figure 4, some embodiments of the present invention employ both decoupled access from execution pipelines and simultaneous multithreading in a unique way. Simultaneous Multithreaded pipelines have been employed in prior art to enhance the utilization of data path units by allowing instructions to be issued from one of several execution threads to each functional unit (e.g., Susan Eggers, University of Wash, papers on Simultaneous Multithreading).

[00127] Decoupled access from execution pipelines have been employed in prior art to enhance the utilization of execution data path units by buffering results from an access unit, which computes addresses to a memory unit that in turn fetches the requested items from memory, and then presenting them to an execution unit (e.g., James E. Smith, paper on Decoupled Access from Execution).

[00128] Compared to conventional pipelines, Eggers prior art used an additional pipeline cycle before instructions could be issued to functional units, the additional cycle needed to determine which threads should be permitted to issue instructions. Consequently, relative to conventional pipelines, the prior art design had additional delay, including dependent branch delay.

[00129] The embodiment shown in Figure 4 contains individual access data path units, with associated register files, for each execution thread. These access units produce addresses, which are aggregated together to a common memory unit, which fetches all the addresses and places the memory contents in one or more buffers. Instructions for execution units, which are shared to varying degrees among the threads are also buffered for later execution. The execution units then perform operations from all active threads using functional data path units that are shared.

[00130] For instructions performed by the execution units, the extra cycle required for prior art simultaneous multithreading designs is overlapped with the memory data access time from prior art decoupled access from execution cycles, so that no additional

delay is incurred by the execution functional units for scheduling resources. For instructions performed by the access units, by employing individual access units for each thread the additional cycle for scheduling shared resources is also eliminated.

[00131] This is a favorable tradeoff because, while threads do not share the access functional units, these units are relatively small compared to the execution functional units, which are shared by threads.

[00132] Figure 12 is a timing diagram of a decoupled pipeline structure in accordance with one embodiment of the present invention. As illustrated in Figure 12, the time permitted by a pipeline to service load operations may be flexibly extended. Here, various types of instructions are abbreviated as A, L, B, E, and S, representing a register-to-register address calculation, a memory load, a branch, a register-to-register data calculation, and a memory store, respectively. According to the present embodiment, the front of the pipeline, in which A, L and B type instructions are handled, is decoupled from the back of the pipeline, in which E, and S type instructions are handled. This decoupling occurs at the point at which the data cache and its backing memory is referenced; similarly, a FIFO that is filled by the instruction fetch unit decouples instruction cache references from the front of the pipeline shown above. The depth of the FIFO structures is implementation-dependent, i.e. not fixed by the architecture. Figure 13 further illustrates this pipeline organization. Accordingly, the latency of load instructions can be hidden, as execute instructions are deferred until the results of the load are available. Nevertheless, the execution unit still processes instructions in normal order, and provides precise exceptions. More details relating to this pipeline structure is explained in the "Superspring Pipeline" section of the Appendix.

[00133] A difficulty in particular pipeline structures is that dependent operations must be separated by the latency of the pipeline, and for highly pipelined machines, the latency of simple operations can be quite significant. According to one embodiment of the present invention, very highly pipelined implementations are provided by alternating execution of two or more independent threads. In an embodiment, a thread is the state

required to maintain an independent execution; the architectural state required is that of the register file contents, program counter, privilege level, local TB, and when required, exception status. In an embodiment, ensuring that only one thread may handle an exception at one time may minimize the latter state, exception status. In order to ensure that all threads make reasonable forward progress, several of the machine resources must be scheduled fairly.

[00134] An example of a resource that is critical that it be fairly shared is the data memory/cache subsystem. In one embodiment, the processor may be able to perform a load operation only on every second cycle, and a store operation only on every fourth cycle. The processor schedules these fixed timing resources fairly by using a round-robin schedule for a number of threads that is relatively prime to the resource reuse rates. In one embodiment, five simultaneous threads of execution ensure that resources which may be used every two or four cycles are fairly shared by allowing the instructions which use those resources to be issued only on every second or fourth issue slot for that thread. More details relating to this pipeline structure are explained in the "Superthread Pipeline" section of the Appendix.

[00135] Referring back to Figure 4, with regard to the sharing of execution units, one embodiment of the present invention employs several different classics of functional units for the execution unit, with varying cost, utilization, and performance. In particular, the G units, which perform simple addition and bitwise operations is relatively inexpensive (in area and power) compared to the other units, and its utilization is relatively high. Consequently, the design employs four such units, where each unit can be shared between two threads. The X unit, which performs a broad class of data switching functions is more expensive and less used, so two units are provided that are each shared among two threads. The T unit, which performs the Wide Translate instruction, is expensive and utilization is low, so the single unit is shared among all four threads. The E unit, which performs the class of Ensemble instructions, is very expensive in area and power compared to the other functional units, but utilization is relatively high, so we provide two such units, each unit shared by two threads.

[00136] In Figure 4, four copies of an access unit are shown, each with an access instruction fetch queue A-Queue 401-404, coupled to an access register file AR 405-408, each of which is, in turn, coupled to two access functional units A 409-416. The access units function independently for four simultaneous threads of execution. These eight access functional units A 409-416 produce results for access register files AR 405-408 and addresses to a shared memory system 417. The memory contents fetched from memory system 417 are combined with execute instructions not performed by the access unit and entered into the four execute instruction queues E-Queue 421-424. Instructions and memory data from E-queue 421-424 are presented to execution register files 425-428, which fetches execution register file source operands. The instructions are coupled to the execution unit arbitration unit Arbitration 431, that selects which instructions from the four threads are to be routed to the available execution units E 441 and 449, X 442 and 448, G 443-444 and 446-447, and T 445. The execution register file source operands ER 425-428 are coupled to the execution units 441-445 using source operand buses 451-454 and to the execution units 445-449 using source operand buses 455-458. The function unit result operands from execution units 441-445 are coupled to the execution register file using result bus 461 and the function units result operands from execution units 445-449 are coupled to the execution register file using result bus 462.

[00137] The foregoing elements of the present invention may be better understood with reference to the attached Appendix.

[00138] In a still further aspect of the present invention, an improved interprivilege gateway is described which involves increased parallelism and leads to enhanced performance. In U.S. Application No. 08/541,416, now U.S. Patent No. 6,101,590, a system and method is described for implementing an instruction that, in a controlled fashion, allows the transfer of control (branch) from a lower-privilege level to a higher-privilege level. Embodiment of the present invention provides an improved system and method for a modified instruction that accomplishes the same purpose but with specific advantages.

[00139] Many processor resources, such as control of the virtual memory system itself, input and output operations, and system control functions are protected from accidental or malicious misuse by enclosing them in a protective, privileged region. Entry to this region must be established only though particular entry points, called gateways, to maintain the integrity of these protected regions.

[00140] Prior art versions of this operation generally load an address from a region of memory using a protected virtual memory attribute that is only set for data regions that contain valid gateway entry points, then perform a branch to an address contained in the contents of memory. Basically, three steps were involved: load, branch, then check. Compared to other instructions, such as register-to-register computation instructions and memory loads and stores, and register-based branches, this is a substantially longer operation, which introduces delays and complexity to a pipelined implementation.

[00141] In the present invention, the branch-gateway instruction performs two operations in parallel: 1) a branch is performed to the contents of register 0 and 2) a load is performed using the contents of register 1, using a specified byte order (little-endian) and a specified size (64 bits). If the value loaded from memory does not equal the contents of register 0, the instruction is aborted due to an exception. In addition, 3) a return address (the next sequential instruction address following the branch-gateway instruction) is written into register 0, provided the instruction is not aborted. This approach essentially uses a first instruction to establish the requisite permission to allow user code to access privileged code, and then a second instruction is permitted to branch directly to the privileged code because of the permissions issued for the first instruction.

[00142] In the present invention, the new privilege level is also contained in register 0, and the second parallel operation does not need to be performed if the new privilege level is not greater than the old privilege level. When this second operation is suppressed, the remainder of the instruction performs an identical function to a branch-link instruction, which is used for invoking procedures that do not require an

increase in privilege. The advantage that this feature brings is that the branch-gateway instruction can be used to call a procedure that may or may not require an increase in privilege.

[00143] The memory load operation verifies with the virtual memory system that the region that is loaded has been tagged as containing valid gateway data. A further advantage of the present invention is that the called procedure may rely on the fact that register 1 contains the address that the gateway data was loaded from, and can use the contents of register 1 to locate additional data or addresses that the procedure may require. Prior art versions of this instruction required that an additional address be loaded from the gateway region of memory in order to initialize that address in a protected manner - the present invention allows the address itself to be loaded with a "normal" load operation that does not require special protection.

[00144] The present invention allows a "normal" load operation to also load the contents of register 0 prior to issuing the branch-gateway instruction. The value may be loaded from the same memory address that is loaded by the branch-gateway instruction, because the present invention contains a virtual memory system in which the region may be enabled for normal load operations as well as the special "gateway" load operation performed by the branch-gateway instruction.

[00145] In a further aspect of the present invention, a system and method is provided for performing a three-input bitwise Boolean operation in a single instruction. A novel method described in detail in appendix is used to encode the eight possible output states of such an operation into only seven bits, and decoding these seven bits back into the eight states.

[00146] In yet a further aspect to the present invention, a system and method is described for improving the branch prediction of simple repetitive loops of code. The method includes providing a count field for indicating how many times a branch is likely to be taken before it is not taken, which enhances the ability to properly predict both the

initial and final branches of simple loops when a compiler can determine the number of iterations that the loop will be performed. This improves performance by avoiding misprediction of the branch at the end of a loop.

Memory Management

[00147] This section discusses the caches, the translation mechanisms, the memory interfaces, and how the multiprocessor interface is used to maintain cache coherence.

a. Overview

[00148] Figure 14 is a diagram illustrating the basic organization of the memory management system according to one embodiment of the invention. In accordance with this embodiment, the processor provides for both local and global virtual addressing, arbitrary page sizes, and coherent-cache multiprocessing. The memory management system is designed to provide the requirements for implementation of virtual machines as well as virtual memory. All facilities of the memory management system are themselves memory mapped, in order to provide for the manipulation of these facilities by high-level language, compiled code. The translation mechanism is designed to allow full byte-at-a-time control of access to the virtual address space, with the assistance of fast exception handlers. Privilege levels provide for the secure transition between insecure user code and secure system facilities. Instructions execute at a privilege, specified by a two-bit field in the access information. Zero is the least-privileged level, and three is the most-privileged level.

[00149] In general terms, the memory management starts from a local virtual address. The local virtual address is translated to a global virtual address by an LTB (Local Translation Buffer). In turn, the global virtual address is translated to a physical address by a GTB (Global Translation Buffer). One of the addresses, a local virtual address, a global virtual address, or a physical address, is used to index the cache data and cache tag arrays, and one of the addresses is used to check the cache tag array for cache presence. Protection information is assembled from the LTB, GTB, and optionally the cache tag, to determine if the access is legal.

[00150] This form varies somewhat, depending on implementation choices made. Because the LTB leaves the lower 48 bits of the address alone, indexing of the cache arrays with the local virtual address is usually identical to cache arrays indexed by the global virtual address. However, indexing cache arrays by the global virtual address rather than the physical address produces a coherence issue if the mapping from global virtual address to physical is many-to-one.

[00151] Starting from a local virtual address, the memory management system performs three actions in parallel: the low-order bits of the virtual address are used to directly access the data in the cache, a low-order bit field is used to access the cache tag, and the high-order bits of the virtual address are translated from a local address space to a global virtual address space.

[00152] Following these three actions, operations vary depending upon the cache implementation. The cache tag may contain either a physical address and access control information (a physically-tagged cache), or may contain a global virtual address and global protection information (a virtually-tagged cache).

[00153] For a physically-tagged cache, the global virtual address is translated to a physical address by the GTB, which generates global protection information. The cache tag is checked against the physical address, to determine a cache hit. In parallel, the local and global protection information is checked.

[00154] For a virtually-tagged cache, the cache tag is checked against the global virtual address, to determine a cache hit, and the local and global protection information is checked. If the cache misses, the global virtual address is translated to a physical address by the GTB, which also generates the global protection information.

b. Local Translation Buffer

[00155] The 64-bit global virtual address space is global among all tasks. In a multitask environment, requirements for a task-local address space arise from operations such as the UNIX "fork" function, in which a task is duplicated into parent and child tasks, each now having a unique virtual address space. In addition, when switching tasks, access to one task's address space must be disabled and another task's access enabled.

[00156] The processor provides for portions of the address space to be made local to individual tasks, with a translation to the global virtual space specified by four 16-bit registers for each local virtual space. The registers specify a mask selecting which of the high-order 16 address bits are checked to match a particular value, and if they match, a value with which to modify the virtual address. The processor avoids setting a fixed page size or local address size; these can be set by software conventions.

[00157] A local virtual address space is specified by the following:

field name	size	description
lm	16	mask to select fields of local virtual address to perform match over
la	16	value to perform match with masked local virtual address
lx	16	value to xor with local virtual address if matched
lp	16	local protection field (detailed later)

local virtual address space specifiers

[00158] There are as many LTB as threads, and up to 23 (8) entries per LTB. Each entry is 128 bits, with the high order 64 bits reserved. Figure 15 illustrates the physical address of an LTB entry for thread th, entry en, byte b.

[00159] Figure 16 illustrates a definition for AccessPhysicalLTB. Figure 17 illustrates how various 16-bit values are packed together into a 64-bit LTB entry. The LTB contains a separate context of register sets for each thread, indicated by the th index above. A context consists of one or more sets of lm/la/lx/lp registers, one set for each simultaneously accessible local virtual address range, indicated by the en index above. This set of registers is called the "Local TB context," or LTB (Local Translation Buffer)

context. The effect of this mechanism is to provide the facilities normally attributed to segmentation. However, in this system there is no extension of the address range, instead, segments are local nicknames for portions of the global virtual address space.

[00160] A failure to match an LTB entry results either in an exception or an access to the global virtual address space, depending on privilege level. A single bit, selected by the privilege level active for the access from a four bit control register field, global access, ga determines the result. If gaPL is zero (0), the failure causes an exception, if it is one (1), the failure causes the address to be directly used as a global virtual address without modification.

[00161] Figure 18 illustrates global access as fields of a control register. Usually, global access is a right conferred to highly privilege levels, so a typical system may be configured with ga0 and ga1 clear (0), but ga2 and ga3 set (1). A single low-privilege (0) task can be safely permitted to have global access, as accesses are further limited by the rwxg privilege fields. A concrete example of this is an emulation task, which may use global addresses to simulate segmentation, such as an x86 emulation. The emulation task then runs as privilege 0, with ga0 set, while most user tasks run as privilege 1, with ga1 clear. Operating system tasks then use privilege 2 and 3 to communicate with and control the user tasks, with ga2 and ga3 set.

[00162] For tasks that have global access disabled at their current privilege level, failure to match an LTB entry causes an exception. The exception handler may load an LTB entry and continue execution, thus providing access to an arbitrary number of local virtual address ranges.

[00163] When failure to match an LTB entry does not cause an exception, instructions may access any region in the local virtual address space, when an LTB entry matches, and may access regions in the global virtual address space when no LTB entry matches. This mechanism permits privileged code to make judicious use of local virtual address ranges, which simplifies the manner in which privileged code may manipulate

the contents of a local virtual address range on behalf of a less-privileged client. Note, however, that under this model, an LTB miss does not cause an exception directly, so the use of more local virtual address ranges than LTB entries requires more care: the local virtual address ranges should be selected so as not to overlap with the global virtual address ranges, and GTB misses to LVA regions must be detected and cause the handler to load an LTB entry.

[00164] Each thread has an independent LTB, so that threads may independently define local translation. The size of the LTB for each thread is implementation dependent and defined as the LE parameter in the architecture description register. LE is the log of the number of entries in the local TB per thread; an implementation may define LE to be a minimum of 0, meaning one LTB entry per thread, or a maximum of 3, meaning eight LTB entries per thread. For the initial Zeus implementation, each thread has two entries and LE=1.

[00165] A minimum implementation of an LTB context is a single set of lm/la/lx/lp registers per thread. However, the need for the LTB to translate both code addresses and data addresses imposes some limits on the use of the LTB in such systems. We need to be able to guarantee forward progress. With a single LTB set per thread, either the code or the data must use global addresses, or both must use the same local address range, as must the LTB and GTB exception handler. To avoid this restriction, the implementation must be raised to two sets per thread, at least one for code and one for data, to guarantee forward progress for arbitrary use of local addresses in the user code (but still be limited to using global addresses for exception handlers).

[00166] As shown in Figure 19, a single-set LTB context may be further simplified by reserving the implementation of the lm and la registers, setting them to a read-only zero value: Note that in such a configuration, only a single LA region can be implemented.

[00167] If the largest possible space is reserved for an address space identifier, the virtual address is partitioned as shown in Figure 20. Any of the bits marked as "local" below may be used as "offset" as desired.

[00168] To improve performance, an implementation may perform the LTB translation on the value of the base register (rc) or unincremented program counter, provided that a check is performed which prohibits changing the unmasked upper 16 bits by the add or increment. If this optimization is provided and the check fails, an AccessDisallowedByVirtualAddress should be signaled. If this optimization is provided, the architecture description parameter LB=1. Otherwise LTB translation is performed on the local address, la, no checking is required, and LB=0.

[00169] As shown in Figure 21, the LTB protect field controls the minimum privilege level required for each memory action of read (r), write (w), execute (x), and gateway (g), as well as memory and cache attributes of write allocate (wa), detail access (da), strong ordering (so), cache disable (cd), and write through (wt). These fields are combined with corresponding bits in the GTB protect field to control these attributes for the mapped memory region.

[00170] The meaning of the fields are given by the following table:

name	size	meaning
g	2	minimum privilege required for gateway access
x	2	minimum privilege required for execute access
w	2	minimum privilege required for write access
r	2	minimum privilege required for read access
0	1	reserved
da	1	detail access
so	1	strong ordering
cc	3	cache control

[00171] Figure 22 illustrates a definition for LocalTranslation.

c. Global Translation Buffer

[00172] Global virtual addresses which fail to be accessed in either the LZC, the MTB, the BTB, or PTB are translated to physical references in a table, here named the "Global Translation Buffer," (GTB).

[00173] Each processor may have one or more GTB's, with each GTB shared by one or more threads. The parameter GT, the base-two log of the number of threads which share a GTB, and the parameter T, the number of threads, allow computation of the number of GTBs (T/2GT), and the number of threads which share each GTB (2GT).

[00174] If there are two GTBs and four threads (GT=1, T=4), GTB 0 services references from threads 0 and 1, and GTB 1 services references from threads 2 and 3. In the first implementation, there is one GTB, shared by all four threads. (GT=2, =4).The GTB has 128 entries (G=7).

[00175] Per clock cycle, each GTB can translate one global virtual address to a physical address, yielding protection information as a side effect.

[00176] A GTB miss causes a software trap. This trap is designed to permit a fast handler for GlobalTBMiss to be written in software, by permitting a second GTB miss to occur as an exception, rather than a machine check.

[00177] There may be as many GTB as threads, and up to 215 entries per GTB. Figure 23 illustrates the physical address of a GTB entry for thread th, entry en, byte b. Note that in Figure 23, the low-order GT bits of the th value are ignored, reflecting that 2GT threads share a single GTB. A single GTB shared between threads appears multiple times in the address space. GTB entries are packed together so that entries in a GTB are consecutive:

[00178] Figure 24 illustrates a definition for AccessPhysicalGTB. Figure 25 illustrates the format of a GTB entry. As shown, each GTB entry is 128 bits.gs = ga + size/2: 256 ≤ size ≤ 264, ga, global address, is aligned (a multiple of) size. px = pa ^ ga. pa, ga, and px are all aligned (a multiple of) size.

[00179] The meaning of the fields are given by the following table:

name	size	meaning
gs	57	global address with size
px	56	physical xor
g	2	minimum privilege required for gateway access
x	2	minimum privilege required for execute access
w	2	minimum privilege required for write access
r	2	minimum privilege required for read access
0	1	reserved
da	1	detail access
so	1	strong ordering
cc	3	cache control

[00180] If the entire contents of the GTB entry is zero (0), the entry will not match any global address at all. If a zero value is written, a zero value is read for the GTB entry. Software must not write a zero value for the gs field unless the entire entry is a zero value.

[00181] It is an error to write GTB entries that multiply match any global address; all GTB entries must have unique, non-overlapping coverage of the global address space. Hardware may produce a machine check if such overlapping coverage is detected, or may produce any physical address and protection information and continue execution.

[00182] Limiting the GTB entry size to 128 bits allows up to replace entries atomically (with a single store operation), which is less complex than the previous design, in which the mask portion was first reduced, then other entries changed, then the mask is expanded. However, it is limiting the amount of attribute information or physical

address range we can specify. Consequently, we are encoding the size as a single additional bit to the global address in order to allow for attribute information. Figure 26 illustrates a definition for GlobalAddressTranslation.

d. GTB Registers

[00183] Because the processor contains multiple threads of execution, even when taking virtual memory exceptions, it is possible for two threads to nearly simultaneously invoke software GTB miss exception handlers for the same memory region. In order to avoid producing improper GTB state in such cases, the GTB includes access facilities for indivisibly checking and then updating the contents of the GTB as a result of a memory write to specific addresses.

[00184] A 128-bit write to the address GTBUpdateFill (fill=1), as a side effect, causes first a check of the global address specified in the data against the GTB. If the global address check results in a match, the data is directed to write on the matching entry. If there is no match, the address specified by GTBLast is used, and GTBLast is incremented. If incrementing GTBLast results in a zero value, GTBLast is reset to GTBFirst, and GTBBump is set. Note that if the size of the updated value is not equal to the size of the matching entry, the global address check may not adequately ensure that no other entries also cover the address range of the updated value. The operation is unpredictable if multiple entries match the global address.

[00185] The GTBUpdateFill register is a 128-bit memory-mapped location, to which a write operation performs the operation defined above. A read operation returns a zero value. The format of the GTBUpdateFill register is identical to that of a GTB entry.

[00186] An alternative write address, GTBUpdate, (fill=0) updates a matching entry, but makes no change to the GTB if no entry matches. This operation can be used to indivisibly update a GTB entry as to protection or physical address information. Figure 27 illustrates a definition for GTBUpdateWrite.

[00187] There may be as many GTB as threads, and up to 211 registers per GTB (5 registers are implemented). Figure 28 illustrates the physical address of a GTB control register for thread th, register rn, byte b. Note that in Figure 28, the low-order GT bits of the th value are ignored, reflecting that 2GT threads share single GTB registers. A single set of GTB registers shared between threads appears multiple times in the address space, and manipulates the GTB of the threads with which the registers are associated.

[00188] The GTBUpdate register is a 128-bit memory-mapped location, to which a write operation performs the operation defined above. A read operation returns a zero value. The format of the GTBUpdateFill register is identical to that of a GTB entry. Figure 29 illustrates the registers GTBLast, GTBFirst, and GTBBump. The registers GTBLast, GTBFirst, and GTBBump are memory mapped. As shown in Figure 29, the GTBLast and GTBFirst registers are G bits wide, and the GTBBump register is one bit.

[00189] Figure 30 illustrates a definition for AccessPhysicalGTBRegisters.

e. Address Generation

[00190] The address units of each of the four threads provide up to two global virtual addresses of load, store, or memory instructions, for a total of eight addresses. LTB units associated with each thread translate the local addresses into global addresses. The LZC operates on global addresses. MTB, BTB, and PTB units associated with each thread translate the global addresses into physical addresses and cache addresses. (A PTB unit associated with each thread produces physical addresses and cache addresses for program counter references. – this is optional, as by limiting address generation to two per thread, the MTB can be used for program references.) Cache addresses are presented to the LOC as required, and physical addresses are checked against cache tags as required.

Rounding and Exceptions

[00191] In accordance with one embodiment of the invention, rounding is specified within the instructions explicitly, to avoid explicit state registers for a rounding

mode. Similarly, the instructions explicitly specify how standard exceptions (invalid operation, division by zero, overflow, underflow and inexact) are to be handled.

[00192] In this embodiment, when no rounding is explicitly named by the instruction (default), round to nearest rounding is performed, and all floating-point exception signals cause the standard-specified default result, rather than a trap. When rounding is explicitly named by the instruction (N: nearest, Z: zero, F: floor, C: ceiling), the specified rounding is performed, and floating-point exception signals other than inexact cause a floating-point exception trap. When X (exact, or exception) is specified, all floating-point exception signals cause a floating-point exception trap, including inexact. More details regarding rounding and exceptions are described in the "Rounding and Exceptions" section of the Appendix.

Group Boolean

[00193] In accordance with one embodiment of the invention, the processor handles a variety Group Boolean operations. For example, Figure 31A presents various Group Boolean instructions. Figures 31B and 31C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the Boolean instructions shown in Figure 31A. As shown in Figures 31B and 31C, in this exemplary embodiment, three values are taken from the contents of registers rd, rc and rb. The ih and il fields specify a function of three bits, producing a single bit result. The specified function is evaluated for each bit position, and the results are catenated and placed in register rd. Register rd is both a source and destination of this instruction.

[00194] The function is specified by eight bits, which give the result for each possible value of the three source bits in each bit position:

d	1 1 1 1 0 0 0 0
c	1 1 0 0 1 1 0 0
b	1 0 1 0 1 0 1 0
f(d,c,b)	f_7 f_6 f_5 f_4 f_3 f_2 f_1 f_0

[00195] A function can be modified by rearranging the bits of the immediate value. The table below shows how rearrangement of immediate value $f_{7..0}$ can reorder the operands d,c,b for the same function.

operation	immediate
f(d,c,b)	f_7 f_6 f_5 f_4 f_3 f_2 f_1 f_0
f(c,d,b)	f_7 f_6 f_3 f_2 f_5 f_4 f_1 f_0
f(d,b,c)	f_7 f_5 f_6 f_4 f_3 f_1 f_2 f_0
f(b,c,d)	f_7 f_3 f_5 f_1 f_6 f_2 f_4 f_0
f(c,b,d)	f_7 f_5 f_3 f_1 f_6 f_4 f_2 f_0
f(b,d,c)	f_7 f_3 f_6 f_2 f_5 f_1 f_4 f_0

[00196] By using such a rearrangement, an operation of the form: *b=f(d,c,b)* can be recoded into a legal form: *b=f(b,d,c)*. For example, the function: *b=f(d,c,b)=d?c:b* cannot be coded, but the equivalent function: *d=c?b:d* can be determined by rearranging the code for *d= f(d,c,b)=d?c:b*, which is 11001010, according to the rule for *f(d,c,b)⇒f(c,b,d)*, to the code 11011000.

[00197] Encoding - Some special characteristics of this rearrangement is the basis of the manner in which the eight function specification bits are compressed to seven immediate bits in this instruction. As seen in the table above, in the general case, a rearrangement of operands from *f(d,c,b)* to *f(d,b,c)*.(interchanging rc and rb) requires interchanging the values of f_6 and f_5 and the values of f_2 and f_1.

[00198] Among the 256 possible functions which this instruction can perform, one quarter of them (64 functions) are unchanged by this rearrangement. These functions have the property that $f_6=f_5$ and $f_2=f_1$. The values of rc and rb can be freely interchanged, and so are sorted into rising or falling order to indicate the value of f_2. These functions are encoded by the values of f_7, f_6, f_4, f_3, and f_0 in the immediate field and f_2 by whether rc>rb, thus using 32 immediate values for 64 functions.

[00199] Another quarter of the functions have f_6=1 and f_5=0. These functions are recoded by interchanging rc and rb, f_6 and f_5, f_2 and f_1. They then share the same encoding as the quarter of the functions where f_6=0 and f_5=1, and are encoded by the values of f_7, f_4, f_3, f_2, f_1, and f_0 in the immediate field, thus using 64 immediate values for 128 functions.

[00200] The remaining quarter of the functions have f_6=f_5 and f_2≠f_1. The half of these in which f_2=1 and f_1=0 are recoded by interchanging rc and rb, f_6 and f_5, f_2 and f_1. They then share the same encoding as the eighth of the functions where f_2=0 and f_1=1, and are encoded by the values of f_7, f_6, f_4, f_3, and f_0 in the immediate field, thus using 32 immediate values for 64 functions.

[00201] The function encoding is summarized by the table:

f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0	trc>trb	ih	il_5	il_4	il_3	il_2	il_1	il_0	rc	rb
		f_6				f_2		f_2	0	0	f_6	f_7	f_4	f_3	f_0	trc	trb
		f_6				f_2		~f_2	0	0	f_6	f_7	f_4	f_3	f_0	trb	trc
		f_6			0	1			0	1	f_6	f_7	f_4	f_3	f_0	trc	trb
		f_6			1	0			0	1	f_6	f_7	f_4	f_3	f_0	trb	trc
	0	1							1	f_2	f_1	f_7	f_4	f_3	f_0	trc	trb
	1	0							1	f_1	f_2	f_7	f_4	f_3	f_0	trb	trc

[00202] The function decoding is summarized by the table:

ih	il_5	il_4	il_3	il_2	il_1	il_0	rc>rb	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0
0	0						0	il_3	il_4	il_4	il_2	il_1	0	0	il_0
0	0						1	il_3	il_4	il_4	il_2	il_1	1	1	il_0
0	1							il_3	il_4	il_4	il_2	il_1	0	1	il_0
1								il_3	0	1	il_2	il_1	il_5	il_4	il_0

Group Multiplex

[00203] In accordance with one embodiment of the invention, the processor handles group multiplex operations. Figures 31D and 31E illustrate an exemplary

embodiment of a format and operation codes that can be used to perform the various Group Multiplex instructions. As shown in Figures 31D and 31E, in this exemplary embodiment, the contents of registers rd, rc and rb are fetched. Each bit of the result is equal to the corresponding bit of rc, if the corresponding bit of rd is set, otherwise it is the corresponding bit of rb. The result is placed into register ra. While the use of three operand registers and a different result register is described here and elsewhere in the present specification, other arrangements, such as the use of immediate values, may also be implemented.

[00204] The table marked Redundancies in Figure 31D illustrates that for particular values of the register specifiers, the Group Multplex operation performs operations otherwise available within the Group Boolean instructions. More specifically, when the result register ra is also present as a source register in the first, second or third source operand position of the operation, the operation is equivalent to the Group Boolean instruction with arguments of 0x11001010, 0x11100010, or 0x11011000 respectively. When the first source operand is the same as the second or third source operand, the Group Multiplex operation is equivalent to a bitwise OR or AND operation respectively.

Group Add

[00205] In accordance with one embodiment of the invention, the processor handles a variety of fixed-point, or integer, group operations. For example, Figure 32A presents various examples of Group Add instructions accommodating different operand sizes, such as a byte (8 bits), doublet (16 bits), quadlet (32 bits), octlet (64 bits), and hexlet (128 bits). Figures 32B and 32C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Group Add instructions shown in Figure 32A. As shown in Figures 32B and 32C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and added, and if specified, checked for overflow or limited, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd. While the use of two operand registers and a different result register is

described here and elsewhere in the present specification, other arrangements, such as the use of immediate values, may also be implemented.

[00206] In the present embodiment, for example, if the operand size specified is a byte (8 bits), and each register is 128-bit wide, then the content of each register may be partitioned into 16 individual operands, and 16 different individual add operations may take place as the result of a single Group Add instruction. Other instructions involving groups of operands may perform group operations in a similar fashion.

Group Subtract

[00207] Similarly, Figure 33A presents various examples of Group Subtract instructions accommodating different operand sizes. Figures 33B and 33C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Group Subtract instructions. As shown in Figures 33B and 33C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and subtracted, and if specified, checked for overflow or limited, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd.

Group Set

[00208] Figure 33A also presents various examples of Group Set instructions accommodating different operand sizes. Figure 33A also presents additional pseudo-instructions which are equivalent to other Group Set instructions according to the mapping rules further presented in Figure 33A. Figures 33B and 33C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Group Set instructions. As shown in Figures 33B and 33C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and the specified comparisons are performed, each producing a Boolean result repeated to the size specified, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd. In the present embodiment, certain comparisons between two identically specified registers, for

which the result of such comparisons would be predictable no matter what the contents of the register, are used to encode comparisons against a zero value.

Combination of Group Set and Boolean operations

[00209] In an embodiment of the invention, conditional operations are provided in the sense that the set on condition operations can be used to construct bit masks that can select between alternate vector expressions, using the bitwise Boolean operations.

Ensemble Divide/Multiply

[00210] Embodiments of the invention provide for other fixed-point group operations also. Figure 34A presents various examples of Ensemble Divide and Ensemble Multiply instructions accommodating different operand sizes. Figures 34B and 34C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Ensemble Divide and Ensemble Multiply instructions. As shown in Figures 34B and 34C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and divided or multiplied, yielding a group of results. The group of results is catenated and placed in register rd.

Group Compare

[00211] Figure 35A present various examples of Group Compare instructions accommodating different operand sizes. Figures 35B and 35C illustrate an exemplary embodiment of a format and operational codes that can be used to perform the various Group Compare instructions. As shown in Figures 35B and 35C, in this exemplary embodiment, these operations perform calculations on partitions of bits in two general register values, and generate a fixed-point arithmetic exception if the condition specified is met. Two values are taken from the contents of registers rd and rc. The specified condition is calculated on partitions of the operands. If the specified condition is true for any partition, a fixed-point arithmetic exception is generated.

Ensemble Unary

[00212] Figure 36A present various examples of Ensemble Unary instructions accommodating different operand sizes. Figures 36B and 36C illustrate an exemplary embodiment of a format and operational codes that can be used to perform the various Ensemble Unary instructions. As shown in Figures 36B and 36C, in this exemplary embodiment, these operations take operands from a register, perform operations on partitions of bits in the operand, and place the concatenated results in a second register. Values are taken from the contents of register rc. The specified operation is performed, and the result is placed in register rd. The code E.SUM.U.1 in Figure 36A is preferably encoded as E.SUM.U.128.

Ensemble Floating-Point Add, Divide, Multiply, and Subtract

[00213] In accordance with one embodiment of the invention, the processor also handles a variety floating-point group operations accommodating different operand sizes. Here, the different operand sizes may represent floating-point operands of different precisions, such as half-precision (16 bits), single-precision (32 bits), double-precision (64 bits), and quad-precision (128 bits). Figure 37 illustrates exemplary functions that are defined for use within the detailed instruction definitions in other sections and figures. In the functions set forth in Figure 37, an internal format represents infinite-precision floating-point values as a four-element structure consisting of (1) s (sign bit): 0 for positive, 1 for negative, (2) t (type): NORM, ZERO, SNAN, QNAN, INFINITY, (3) e (exponent), and (4) f: (fraction). The mathematical interpretation of a normal value places the binary point at the units of the fraction, adjusted by the exponent: (-1)^s*(2^e)*f. The function F converts a packed IEEE floating-point value into internal format. The function PackF converts an internal format back into IEEE floating-point format, with rounding and exception control.

[00214] Figures 38A and 39A present various examples of Ensemble Floating Point Add, Divide, Multiply, and Subtract instructions. Figures 38B-C and 39B-C illustrate an exemplary embodiment of formats and operation codes that can be used to

perform the various Ensemble Floating Point Add, Divide, Multiply, and Subtract instructions. In these examples, Ensemble Floating Point Add, Divide, and Multiply instructions have been labeled as "EnsembleFloatingPoint." Also, Ensemble Floating-Point Subtract instructions have been labeled as "EnsembleReversedFloatingPoint." As shown in Figures 38B-C and 39B-C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified, and the specified group operation is performed, yielding a group of results. The group of results is catenated and placed in register rd.

[00215] In the present embodiment, the operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Ensemble Multiply-Add Floating-Point

[00216] Figure 38D present various examples of Ensemble Floating Point Multiply Add instructions. Figures 38E-F illustrate an exemplary embodiment of formats and operation codes that can be used to perform the various Ensemble Floating Point Multiply Add instructions. In these examples, Ensemble Floating Point Multiply Add instructions have been labeled as "EnsembleInplaceFloatingPoint." As shown in Figures 38E-F, in this exemplary embodiment, operations take operands from three registers, perform operations on partitions of bits in the operands, and place the concatenated results in the third register. Specifically, the contents of registers rd, rc and rb are partitioned into groups of operands of the size specified, and for each partitioned element, the contents of registers rc and rb are multiplied and added to the contents of register rd, yielding a group of results. The group of results is catenated and placed in register rd. Register rd is both a source and destination of this instruction.

[00217] In the present embodiment, the operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Group Scale-Add Floating-Point

[00218] Figure 38G present various examples of Ensemble Floating Point Scale Add instructions. Figures 38H-I illustrate an exemplary embodiment of formats and operation codes that can be used to perform the various Ensemble Floating Point Scale Add instructions. In these examples, Ensemble Floating Point Scale Add instructions have been labeled as "EnsembleTernaryFloatingPoint." As shown in Figures 38E-F, in this exemplary embodiment, the contents of registers rd and rc are taken to represent a group of floating-point operands. Operands from register rd are multiplied with a floating-point operand taken from the least-significant bits of the contents of register rb and added to operands from register rc multiplied with a floating-point operand taken from the next least-significant bits of the contents of register rb. The results are concatenated and placed in register ra. In an exemplary embodiment, the results are rounded to the nearest representable floating-point value in a single floating-point operation. In an exemplary embodiment, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754. In an exemplary embodiment, these instructions cannot select a directed rounding mode or trap on inexact.

Group Set Floating-Point

[00219] Figure 39D also presents various examples of Group Set Floating-point instructions accommodating different operand sizes. Figure 39E also presents additional pseudo-instructions which are equivalent to other Group Set Floating-Point instructions according to the mapping rules further presented in Figure 39E. Figures 39F and 39G illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Group Set instructions. As shown in Figures 39G, in this exemplary

embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and the specified comparisons are performed, each producing a Boolean result repeated to the size specified, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd. If a rounding mode is specified a floating-point exception is raised if any operand is a SNAN, or when performing a Less or Greater Equal comparison, any operand is a QNAN. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Group Compare Floating-point

[00220] Figure 40A present various examples of Group Compare Floating-point instructions accommodating different operand sizes. Figures 40B and 40C illustrate an exemplary embodiment of a format and operational codes that can be used to perform the various Group Compare Floating-point instructions. As shown in Figures 40B and 40C, in this exemplary embodiment, these operations perform calculations on partitions of bits in two general register values, and generate a floating-point arithmetic exception if the condition specified is met. The contents of registers rd and rc are compared using the specified floating-point condition. If the result of the comparison is true for any corresponding pair of elements, a floating-point exception is raised. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation occurs. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Ensemble Unary Floating-point

[00221] Figure 41A present various examples of Ensemble Unary Floating-point instructions accommodating different operand sizes. Figures 41B and 41C illustrate an exemplary embodiment of a format and operational codes that can be used to perform the various Ensemble Unary Floating-point instructions. As shown in Figures 41B and 41C, in this exemplary embodiment, these operations take one value from a register, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the concatenated results in a register. The contents of register rc is used as the

operand of the specified floating-point operation. The result is placed in register rd. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, unless default exception handling is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified or if default exception handling is specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754. The reciprocal estimate and reciprocal square root estimate instructions compute an exact result for half precision, and a result with at least 12 bits of significant precision for larger formats.

Ensemble Multiply Galois Field

[00222] In accordance with one embodiment of the invention, the processor handles different Galois filed operations. For example, Figure 42A presents various examples of Ensemble Multiply Gaois Field instructions accommodating different operand sizes. Figures 42B and 42C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the Ensemble Multiply Gaois Field instructions shown in Figure 42A. As shown in Figures 42B and 32C, in this exemplary embodiment, the contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

[00223] The contents of registers rd and rc are partitioned into groups of operands of the size specified and multiplied in the manner of polynomials. The group of values is reduced modulo the polynomial specified by the contents of register rb, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register ra.

[00224] An ensemble multiply Galois field bytes instruction (E.MULG.8) multiplies operand [d15 d14 d13 d12 d11 d10 d9 d8 d7 d6 d5 d4 d3 d2 d1 d0] by operand [c15 c14 c13 c12 c11 c10 c9 c8 c7 c6 c5 c4 c3 c2 c1 c0], modulo polynomial [q],

yielding the results [(d15c15 mod q) (d14c14 mod q) ... (d0c0 mod q), as illustrated in Figure 42D.

Compress, Expand, Rotate and Shift

[00225] In one embodiment of the invention, crossbar switch units such as units 142 and 148 perform data handling operations, as previously discussed. As shown in Figure 43A, such data handling operations may include various examples of Crossbar Compress, Crossbar Expand, Crossbar Rotate, and Crossbar Shift operations. Figures 43B and 43C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Crossbar Compress, Crossbar Expand, Crossbar Rotate, and Crossbar Shift instructions. As shown in Figures 43B and 43C, in this exemplary embodiment, the contents of registers rc and rb are obtained and the contents of register rc is partitioned into groups of operands of the size specified and the specified operation is performed using a shift amount obtained from the contents of register rb masked to values from zero to one less than the size specified, yielding a group of results. The group of results is catenated and placed in register rd.

[00226] Various Group Compress operations may convert groups of operands from higher precision data to lower precision data. An arbitrary half-sized sub-field of each bit field can be selected to appear in the result. For example, Figure 43D shows an X.COMPRESS.16 rd=rc, 4 operation, which performs a selection of bits 19..4 of each quadlet in a hexlet.Various Group Shift operations may allow shifting of groups of operands by a specified number of bits, in a specified direction, such as shift right or shift left. As can be seen in Figure 43C, certain Group Shift Left instructions may also involve clearing (to zero) empty low order bits associated with the shift, for each operand. Certain Group Shift Right instructions may involve clearing (to zero) empty high order bits associated with the shift, for each operand. Further, certain Group Shift Right instructions may involve filling empty high order bits associated with the shift with copies of the sign bit, for each operand.

Shift Merge

[00227] In one embodiment of the invention, as shown in Figure 43E, such data handling operations may also include various examples of Shift Merge operations. Figures 43F and 43G illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Shift Merge instructions. As shown in Figures 43F and 43G, in this exemplary embodiment, the contents of registers rd, and rc are obtained and the contents of register rd and rc are partitioned into groups of operands of the size specified, and the specified operation is performed using a shift amount obtained from the contents of register rb masked to values from zero to one less than the size specified, yielding a group of results. The group of results is catenated and placed in register rd. Register rd is both a source and destination of this instruction.

[00228] Shift Merge operations may allow shifting of groups of operands by a specified number of bits, in a specified direction, such as shift right or shift left. As can be seen in Figure 43G, certain Shift Merge operations may involve filling empty bits associated with the shift with copies of corresponding bits from the contents of register rd, for each operand.

Compress, Expand, Rotate and Shift Immediate

[00229] In one embodiment of the invention, crossbar switch units such as units 142 and 148 perform data handling operations, as previously discussed. As shown in Figure 43H, such data handling operations may include various examples of Crossbar Compress Immediate, Crossbar Expand Immediate, Crossbar Rotate Immediate, and Crossbar Shift Immediate operations. Figures 43I and 43J illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Crossbar Compress Immediate, Crossbar Expand Immediate, Crossbar Rotate Immediate, and Crossbar Shift Immediate instructions. As shown in Figures 43I and 43J, in this exemplary embodiment, the contents of register rc is obtained and is partitioned into groups of operands of the size specified and the specified operation is performed using a shift amount obtained from the instruction masked to values from zero to one less than

the size specified, yielding a group of results. The group of results is catenated and placed in register rd.

[00230] Various Group Compress Immediate operations may convert groups of operands from higher precision data to lower precision data. An arbitrary half-sized sub-field of each bit field can be selected to appear in the result. For example, Figure 43D shows an X.COMPRESS.16 rd=rc,4 operation, which performs a selection of bits 19..4 of each quadlet in a hexlet. Various Group Shift Immediate operations may allow shifting of groups of operands by a specified number of bits, in a specified direction, such as shift right or shift left. As can be seen in Figure 43J, certain Group Shift Left Immediate instructions may also involve clearing (to zero) empty low order bits associated with the shift, for each operand. Certain Group Shift Right Immediate instructions may involve clearing (to zero) empty high order bits associated with the shift, for each operand. Further, certain Group Shift Right Immediate instructions may involve filling empty high order bits associated with the shift with copies of the sign bit, for each operand.

Shift Merge Immediate

[00231] In one embodiment of the invention, as shown in Figure 43K, such data handling operations may also include various examples of Shift Merge Immediate operations. Figures 43L and 43M illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Shift Merge Immediate instructions. As shown in Figures 43L and 43M, in this exemplary embodiment, the contents of registers rd and rc are obtained and are partitioned into groups of operands of the size specified, and the specified operation is performed using a shift amount obtained from the instruction masked to values from zero to one less than the size specified, yielding a group of results. The group of results is catenated and placed in register rd. Register rd is both a source and destination of this instruction.

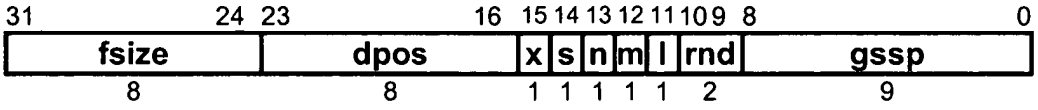
[00232] Shift Merge operations may allow shifting of groups of operands by a specified number of bits, in a specified direction, such as shift right or shift left. As can be seen in Figure 43G, certain Shift Merge operations may involve filling empty bits

associated with the shift with copies of corresponding bits from the contents of register rd, for each operand.

Crossbar Extract

[00233] In one embodiment of the invention, data handling operations may also include a Crossbar Extract instruction. Figures 44A and 44B illustrate an exemplary embodiment of a format and operation codes that can be used to perform the Crossbar Extract instruction. As shown in Figures 44A and 44B, in this exemplary embodiment, the contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

[00234] The Crossbar Extract instruction allows bits to be extracted from different operands in various ways. Specifically, bits 31..0 of the contents of register rb specifies several parameters which control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI.128 instruction (see the Appendix). The control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.:



[00235] The table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	reserved
m	1	merge vs. extract
l	1	reserved
rnd	2	reserved
gssp	9	group size and source position

[00236] The 9-bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512-4***gsize**+**spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**)-1.

[00237] The values in the **s**, **n**, **m**, **l**, and **rnd** fields have the following meaning:

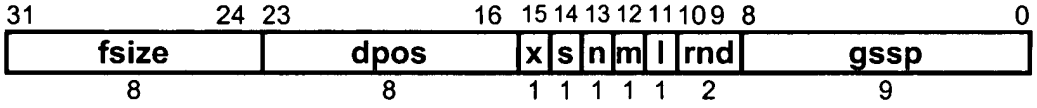
values	s	n	m	l	rnd
0	unsigned		extract		
1	signed		merge		
2					
3					

[00238] As shown in Figure 44C, for the X.EXTRACT instruction, when m=0, the parameters are interpreted to select a fields from the catenated contents of registers rd and rc, extracting values which are catenated and placed in register ra. As shown in Figure 44D, for a crossbar-merge-extract (X.EXTRACT when m=1), the parameters are interpreted to merge fields from the contents of register rd with the contents of register rc. The results are catenated and placed in register ra.

Ensemble Extract

[00239] In one embodiment of the invention, data handling operations may also include an Ensemble Extract instruction. Figures 44E, 44F and 44G illustrate an exemplary embodiment of a format and operation codes that can be used to perform the Ensemble Extract instruction. As shown in Figures 44F and 44G, in this exemplary embodiment, the contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

[00240] The Crossbar Extract instruction allows bits to be extracted from different operands in various ways. Specifically, bits 31..0 of the contents of register rb specifies several parameters which control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI.128 instruction (see the Appendix). The control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.:



[00241] The table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	complex vs. real multiplication
m	1	merge vs. extract or mixed-sign vs. same-sign multiplication
l	1	limit: saturation vs. truncation
rnd	2	rounding
gssp	9	group size and source position

[00242] The 9-bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512-4***gsize**+**spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**)-1.

[00243] The values in the **s**, **n**, **m**, **l**, and **rnd** fields have the following meaning:

values	s	n	m	l	rnd
0	unsigned	real	extract/same-sign	truncate	F
1	signed	complex	merge/mixed-sign	saturate	Z
2					N
3					C

[00244] As shown in Figure 44C, for the E.EXTRACT instruction, when m=0, the parameters are interpreted to select a fields from the catenated contents of registers rd and rc, extracting values which are catenated and placed in register ra. As shown in Figure 44D, for an ensemble-merge-extract (E.EXTRACT when m=1), the parameters are interpreted to merge fields from the contents of register rd with the contents of register rc. The results are catenated and placed in register ra. As can be seen from Figure 44G, the operand portion to the left of the selected field is treated as signed or unsigned as controlled by the **s** field, and truncated or saturated as controlled by the **t** field, while the operand portion to the right of the selected field is rounded as controlled by the **rnd** field.

Deposit and Withdraw

[00245] As shown in Figure 45A, in one embodiment of the invention, data handling operations include various Deposit and Withdraw instructions. Figures 45B and 45C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Deposit and Withdraw instructions. As shown in Figures 45B and 45C, in this exemplary embodiment, these operations take operands from a register and two immediate values, perform operations on partitions of bits in the operands, and place the concatenated results in the second register. Specifically, the contents of register rc is fetched, and 7-bit immediate values are taken from the 2-bit ih and the 6-bit gsfp and gsfs

fields. The specified operation is performed on these operands. The result is placed into register rd.

[00246] Figure 45D shows legal values for the ih, gsfp and gsfs fields, indicating the group size to which they apply. The ih, gsfp and gsfs fields encode three values: the group size, the field size, and a shift amount. The shift amount can also be considered to be the source bit field position for group-withdraw instructions or the destination bit field position for group-deposit instructions. The encoding is designed so that combining the gsfp and gsfs fields with a bitwise-and produces a result which can be decoded to the group size, and so the field size and shift amount can be easily decoded once the group size has been determined.

[00247] As shown in Figure 45E, the crossbar-deposit instructions deposit a bit field from the lower bits of each group partition of the source to a specified bit position in the result. The value is either sign-extended or zero-extended, as specified. As shown in Figure 45F, the crossbar-withdraw instructions withdraw a bit field from a specified bit position in the each group partition of the source and place it in the lower bits in the result. The value is either sign-extended or zero-extended, as specified.

Deposit Merge

[00248] As shown in Figure 45G, in one embodiment of the invention, data handling operations include various Deposit Merge instructions. Figures 45H and 45I illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Deposit Merge instructions. As shown in Figures 45H and 45I, in this exemplary embodiment, these operations take operands from two registers and two immediate values, perform operations on partitions of bits in the operands, and place the concatenated results in the second register. Specifically, the contents of registers rc and rd are fetched, and 7-bit immediate values are taken from the 2-bit ih and the 6-bit gsfp and gsfs fields. The specified operation is performed on these operands. The result is placed into register rd.

[00249] Figure 45D shows legal values for the ih, gsfp and gsfs fields, indicating the group size to which they apply. The ih, gsfp and gsfs fields encode three values: the group size, the field size, and a shift amount. The shift amount can also be considered to be the source bit field position for group-withdraw instructions or the destination bit field position for group-deposit instructions. The encoding is designed so that combining the gsfp and gsfs fields with a bitwise-and produces a result which can be decoded to the group size, and so the field size and shift amount can be easily decoded once the group size has been determined.

[00250] As shown in Figure 45J, the crossbar-deposit-merge instructions deposit a bit field from the lower bits of each group partition of the source to a specified bit position in the result. The value is merged with the contents of register rd at bit positions above and below the deposited bit field. No sign- or zero-extension is performed by this instruction.

Shuffle

[00251] As shown in Figure 46A, in one embodiment of the invention, data handling operations may also include various Shuffle instructions, which allow the contents of registers to be partitioned into groups of operands and interleaved in a variety of ways. Figures 46B and 46C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Shuffle instructions. As shown in Figures 46B and 46C, in this exemplary embodiment, one of two operations is performed, depending on whether the **rc** and **rb** fields are equal. Also, Figure 46B and the description below illustrate the format of and relationship of the **rd**, **rc**, **rb**, **op**, **v**, **w**, **h**, and **size** fields.

[00252] In the present embodiment, if the **rc** and **rb** fields are equal, a 128-bit operand is taken from the contents of register **rc**. Items of size **v** are divided into **w** piles and shuffled together, within groups of **size** bits, according to the value of **op**. The result is placed in register **rd**.

[00253] Figure 46C illustrates that for this operation, values of three parameters **x**, **y**, and **z** are computed depending on the value of **op**, and in each result bit position **i**, a source bit position within the contents of register rc is selected, wherein the source bit position is the catenation of four fields, the first and fourth fields containing fields of **i** which are unchanged: **6..x** and **y-1..0**, and the second and third fields containing a subfield of **i**, bits **x-1..y** which is rotated by an amount **z**: **y+z-1..y** and **x-1..y+z**.

[00254] Further, if the **rc** and **rb** fields are not equal, the contents of registers **rc** and **rb** are catenated into a 256-bit operand. Items of size **v** are divided into **w** piles and shuffled together, according to the value of **op**. Depending on the value of **h**, a sub-field of **op**, the low 128 bits (h=0), or the high 128 bits (h=1) of the 256-bit shuffled contents are selected as the result. The result is placed in register **rd**.

[00255] Figure 46C illustrates that for this operation, the value of **x** is fixed, and values of two parameters **y** and **z** are computed depending on the value of **op**, and in each result bit position **i**, a source bit position within the contents of register rc is selected, wherein the source bit position is the catenation of three fields, the first field containing a fields of **i** which is unchanged: **y-1..0**, and the second and third fields containing a subfield of **i**, bits **x-1..y** which is rotated by an amount **z**: **y+z-1..y** and **x-1..y+z**.

[00256] As shown in Figure 46D, an example of a crossbar 4-way shuffle of bytes within hexlet instruction (X.SHUFFLE.128 rd=rcb,8,4) may divide the 128-bit operand into 16 bytes and partitions the bytes 4 ways (indicated by varying shade in the diagram below). The 4 partitions are perfectly shuffled, producing a 128-bit result. As shown in Figure 46E, an example of a crossbar 4-way shuffle of bytes within triclet instruction (X.SHUFFLE.256 rd=rc,rb,8,4,0) may catenate the contents of rc and rb, then divides the 256-bit content into 32 bytes and partitions the bytes 4 ways (indicated by varying shade in the diagram below). The low-order halves of the 4 partitions are perfectly shuffled, producing a 128-bit result.

[00257] Changing the last immediate value **h** to 1 (X.SHUFFLE.256 rd=rc,rb,8,4,1) may modify the operation to perform the same function on the high-order halves of the 4 partitions. When **rc** and **rb** are equal, the table below shows the value of the **op** field and associated values for **size**, **v**, and **w**.

op	size	v	w
0	4	1	2
1	8	1	2
2	8	2	2
3	8	1	4
4	16	1	2
5	16	2	2
6	16	4	2
7	16	1	4
8	16	2	4
9	16	1	8
10	32	1	2
11	32	2	2
12	32	4	2
13	32	8	2
14	32	1	4
15	32	2	4
16	32	4	4
17	32	1	8
18	32	2	8
19	32	1	16
20	64	1	2
21	64	2	2
22	64	4	2
23	64	8	2
24	64	16	2
25	64	1	4
26	64	2	4
27	64	4	4

op	size	v	w
28	64	8	4
29	64	1	8
30	64	2	8
31	64	4	8
32	64	1	16
33	64	2	16
34	64	1	32
35	128	1	2
36	128	2	2
37	128	4	2
38	128	8	2
39	128	16	2
40	128	32	2
41	128	1	4
42	128	2	4
43	128	4	4
44	128	8	4
45	128	16	4
46	128	1	8
47	128	2	8
48	128	4	8
49	128	8	8
50	128	1	16
51	128	2	16
52	128	4	16
53	128	1	32
54	128	2	32
55	128	1	64

[00258] When **rc** and **rb** are not equal, the table below shows the value of the $\mathbf{op}_{4..0}$ field and associated values for **size**, **v**, and **w**: $\mathbf{Op}_5$ is the value of **h**, which controls whether the low-order or high-order half of each partition is shuffled into the result.

op4..0	size	v	w
0	256	1	2
1	256	2	2
2	256	4	2
3	256	8	2
4	256	16	2
5	256	32	2
6	256	64	2
7	256	1	4
8	256	2	4
9	256	4	4
10	256	8	4
11	256	16	4
12	256	32	4
13	256	1	8
14	256	2	8
15	256	4	8
16	256	8	8
17	256	16	8
18	256	1	16
19	256	2	16
20	256	4	16
21	256	8	16
22	256	1	32
23	256	2	32
24	256	4	32
25	256	1	64
26	256	2	64
27	256	1	128

Swizzle

[00259] In one embodiment of the invention, data handling operations may also include various Crossbar Swizzle instruction. Figures 47A and 47B illustrate an exemplary embodiment of a format and operation codes that can be used to perform Crossbar Swizzle instructions. As shown in Figures 47A and 47B, in this exemplary embodiment, the contents of register rc are fetched, and 7-bit immediate values, icopy and iswap, are constructed from the 2-bit ih field and from the 6-bit icopya and iswapa fields. The specified operation is performed on these operands. The result is placed into register rd.

[00260] The "swizzle" operation can reverse the order of the bit fields in a hexlet. For example, a X.SWIZZLE rd=rc,127,112 operation reverses the doublets within a hexlet, as shown in Figure 47C. In some cases, it is desirable to use a group instruction in which one or more operands is a single value, not an array. The "swizzle" operation can also copy operands to multiple locations within a hexlet. For example, a X.SWIZZLE 15,0 operation copies the low-order 16 bits to each double within a hexlet.

Select

[00261] In one embodiment of the invention, data handling operations may also include various Crossbar Select instruction. Figures 47D and 47E illustrate an exemplary embodiment of a format and operation codes that can be used to perform Crossbar Select instructions. As shown in Figures 47D and 47E, in this exemplary embodiment, the contents of registers rd, rc and rb are fetched, and the contents of registers rd and rc are catenated, producing catenated data dc. The contents of register rb is partitioned into elements, and the value expressed in each partition is employed to select one partitioned element of the catenated data dc. The selected elements are catenated together, and result is placed into register ra.

Bus Interface

[00262] According to one embodiment of the invention, an initial implementation of the processor uses a "Super Socket 7 compatible" (SS7) bus interface, which is generally similar to and compatible with other "Socket 7" and "Super Socket 7" processors. Figure 48 is a pin summary describing the functions of various pins in accordance with the present embodiment. Figures 49A-G contain electrical specifications describing AC and DC parameters in accordance with the present embodiment. Further details are provided in the "Bus Interface" section of the Appendix.

Load and Load Immediate

[00263] As shown in Figure 50A and 51A, in one embodiment of the invention, memory access operations may also include various Load and Load Immediate instructions. These figures and figures 50B and 51B show that the various Load and

Load Immediate instructions specify a type of operand, either signed, or unsigned, represented by omitting or including a U, respectively. The instructions further specify a size of memory operand, byte, double, quadlet, octlet, or hexlet, representing 8, 16, 32, 64, and 128 bits respectively. The instructions further specify aligned memory operands, or not, represented by including a A, or with the A omitted, respectively. The instructions further specify a byte-ordering of the memory operand, either big-endian, or little-endian, represented by B, and L respectively.

[00264] Each instruction specifies the above items with the following exceptions: L.8, L.U8, L.I.8, L.I.U8 need not distinguish between little-endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is loaded. L.128.B, L.128.AB, L.128.L, L.128AL, L.I.128.B, L.I.128.AB, L.I.128.L, and L.I.128AL need not distinguish between signed and unsigned, as the hexlet fills the destination register.

[00265] Figures 50B and 50C illustrate an exemplary embodiment of formats and operation codes that can be used to perform Load instructions. As shown in Figures 50B and 50C, in this exemplary embodiment, an operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the sign-extended value of the offset field, multiplied by the operand size.

[00266] Figures 51B and 51C illustrate an exemplary embodiment of formats and operation codes that can be used to perform Load Immediate instructions. As shown in Figures 51B and 51C, in this exemplary embodiment, an operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the sign-extended value of the offset field, multiplied by the operand size.

[00267] In an exemplary embodiment, for both Load and Load Immediate instructions, the contents of memory using the specified byte order are read, treated as the size specified, zero-extended or sign-extended as specified, and placed into register rd. If alignment is specified, the computed virtual address must be aligned, that is, it must be an

exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Store and Store Immediate

[00268] As shown in Figure 52A and 53A, in one embodiment of the invention, memory access operations may also include various Store and Store Immediate instructions. These figures and figures 52B and 53B show that the various Store and Store Immediate instructions specify a size of memory operand, byte, double, quadlet, octlet, or hexlet, representing 8, 16, 32, 64, and 128 bits respectively. The instructions further specify aligned memory operands, or not, represented by including a A, or with the A omitted, respectively. The instructions further specify a byte-ordering of the memory operand, either big-endian, or little-endian, represented by B, and L respectively.

[00269] Each instruction specifies the above items with the following exceptions: L.8 and L.I.8 need not distinguish between little-endian and big-endian ordering, nor between aligned and unaligned, as only a single byte is stored.

[00270] Figures 52B and 52C illustrate an exemplary embodiment of formats and operation codes that can be used to perform Store instructions. As shown in Figures 52B and 52C, in this exemplary embodiment, an operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the contents of register rb multiplied by operand size.

[00271] Figures 53B and 53C illustrate an exemplary embodiment of formats and operation codes that can be used to perform Store Immediate instructions. As shown in Figures 53B and 53C, in this exemplary embodiment, an operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the sign-extended value of the offset field, multiplied by the operand size.

[00272] In an exemplary embodiment, for both Store and Store Immediate instructions, the contents of register rd, treated as the size specified, is stored in memory using the specified byte order. If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Store Multiplex and Store Multiplex Immediate

[00273] As shown in Figure 52A and 53A, in one embodiment of the invention, memory access operations may also include various Store Multiplex and Store Multiplex Immediate instructions. These figures and figures 52B and 53B show that the various Store Multiplex and Store Multiplex Immediate instructions specify a size of memory operand, octlet, representing 64 bits. The instructions further specify aligned memory operands, represented by including a A. The instructions further specify a byte-ordering of the memory operand, either big-endian, or little-endian, represented by B, and L respectively.

[00274] Figures 52B and 52C illustrate an exemplary embodiment of formats and operation codes that can be used to perform Store Multiplex instructions. As shown in Figures 52B and 52C, in this exemplary embodiment, an operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the contents of register rb multiplied by operand size.

[00275] Figures 53B and 53C illustrate an exemplary embodiment of formats and operation codes that can be used to perform Store Multiplex Immediate instructions. As shown in Figures 53B and 53C, in this exemplary embodiment, an operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the sign-extended value of the offset field, multiplied by the operand size.

[00276] In an exemplary embodiment, for both Store Multiplex and Store Multiplex Immediate instructions, data contents and mask contents of the contents of

register rd are identified. The data contents are stored in memory using the specified byte order for values in which the corresponding mask contents are set. In an exemplary embodiment, it can be understood that masked writing of data can be accomplished by indivisibly reading the original contents of the addressed memory operand, modifying the value, and writing the modified value back to the addressed memory operand. In an exemplary embodiment, the modification of the value is accomplished using an operation previously identified as a Multiplex operation in the section titled Group Multiplex, above, and in Figure 31E.

[00277] In an exemplary embodiment, for both Store Multiplex and Store Multiplex Immediate instructions, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Additional Load and Execute Resources

[00278] In an exemplary embodiment, studies of the dynamic distribution of instructions on various benchmark suites indicate that the most frequently-issued instruction classes are load instructions and execute instructions. In an exemplary embodiment, it is advantageous to consider execution pipelines in which the ability to target the machine resources toward issuing load and execute instructions is increased.

[00279] In an exemplary embodiment, one of the means to increase the ability to issue execute-class instructions is to provide the means to issue two execute instructions in a single-issue string. The execution unit actually requires several distinct resources, so by partitioning these resources, the issue capability can be increased without increasing the number of functional units, other than the increased register file read and write ports. In an exemplary embodiment, the partitioning favored places all instructions that involve shifting and shuffling in one execution unit, and all instructions that involve multiplication, including fixed-point and floating-point multiply and add in another unit. In an exemplary embodiment, resources used for implementing add, subtract, and bitwise logical operations may be duplicated, being modest in size compared to the shift and

multiply units. In another exemplary embodiment, resources used are shared between the two units, as the operations have low-enough latency that two operations might be pipelined within a single issue cycle. These instructions must generally be independent, except in another exemplary embodiment that two simple add, subtract, or bitwise logical instructions may be performed dependently, if the resources for executing simple instructions are shared between the execution units.

[00280] In an exemplary embodiment, one of the means to increase the ability to issue load-class instructions is to provide the means to issue two load instructions in a single-issue string. This would generally increase the resources required of the data fetch unit and the data cache, but a compensating solution is to steal the resources for the store instruction to execute the second load instruction. Thus, in an exemplary embodiment, a single-issue string can then contain either two load instructions, or one load instruction and one store instruction, which uses the same register read ports and address computation resources as the basic 5-instruction string in another exemplary embodiment.

[00281] In an exemplary embodiment, this capability also may be employed to provide support for unaligned load and store instructions, where a single-issue string may contain as an alternative a single unaligned load or store instruction which uses the resources of the two load-class units in concert to accomplish the unaligned memory operation.

High-level language accessibility

[00282] In one embodiment of the invention, all processor, memory, and interface resources directly accessible to high-level language programs. In one embodiment, memory is byte-addressed, using either little-endian or big-endian byte ordering. In one embodiment, for consistency with the bit ordering, and for compatibility with x86 processors, little-endian byte ordering is used when an ordering must be selected. In one embodiment, load and store instructions are available for both little-endian and big-endian byte ordering. In one embodiment, interface resources are accessible as memory-

mapped registers. In one embodiment, system state is memory mapped, so that it can be manipulated by compiled code.

[00283] In one embodiment, instructions are specified to assemblers and other code tools in the syntax of an instruction mnemonic (operation code), then optionally white space followed by a list of operands. In one embodiment, instruction mnemonics listed in this specification are in upper case (capital) letters, assemblers accept either upper case or lower case letters in the instruction mnemonics. In this specification, instruction mnemonics contain periods (".") to separate elements to make them easier to understand; assemblers ignore periods within instruction mnemonics.

[00284] In Figures 31B, 31D, 32B, 33B, 34B, 35B, 36B, 38B, 38E, 38H, 39B, 39F, 40B, 41B, 42B, 43B, 43F, 43I, 43L, 44A, 44F, 45B, 45H, 46B, 47A, 47D, 50B, 51B, 52B, and 53B, the format of instructions to be presented to an assembler is illustrated. Following the assembler format, the format for inclusion of instructions into high-level compiled languages is indicated. Finally, the detailed structure of the instruction fields, including pseudo code used to connect the assembler and compiled formats to the instruction fields is shown. Further detailed explanation of the formats and instruction decoding is provided in the Appendix, in the section titled "Instruction Set."

[00285] In one embodiment, an instruction is specifically defined as a four-byte structure with the little-endian ordering. In one embodiment, instructions must be aligned on four-byte boundaries. In one embodiment, basic floating-point operations supported in hardware are floating-point add, subtract, multiply, divide, square root and conversions among floating-point formats and between floating-point and binary integer formats. Software libraries provide other operations required by the ANSI/IEEE floating-point standard.

[00286] In one embodiment, software conventions are employed at software module boundaries, in order to permit the combination of separately compiled code and to provide standard interfaces between application, library and system software. In one

embodiment, register usage and procedure call conventions may be modified, simplified or optimized when a single compilation encloses procedures within a compilation unit so that the procedures have no external interfaces. For example, internal procedures may permit a greater number of register-passed parameters, or have registers allocated to avoid the need to save registers at procedure boundaries, or may use a single stack or data pointer allocation to suffice for more than one level of procedure call.

[00287] In one embodiment, at a procedure call boundary, registers are saved either by the caller or callee procedure, which provides a mechanism for leaf procedures to avoid needing to save registers. Compilers may choose to allocate variables into caller or callee saved registers depending on how their lifetimes overlap with procedure calls.

[00288] In one embodiment, procedure parameters are normally allocated in registers, starting from register 2 up to register 9. These registers hold up to 8 parameters, which may each be of any size from one byte to sixteen bytes (hexlet), including floating-point and small structure parameters. Additional parameters are passed in memory, allocated on the stack. For C procedures which use varargs.h or stdarg.h and pass parameters to further procedures, the compilers must leave room in the stack memory allocation to save registers 2 through 9 into memory contiguously with the additional stack memory parameters, so that procedures such as _doprnt can refer to the parameters as an array. Procedure return values are also allocated in registers, starting from register 2 up to register 9. Larger values are passed in memory, allocated on the stack.

[00289] In one embodiment, instruction scheduling is performed by a compiler. In the manner of software pipelineing, instructions should generally be scheduled so that previous operations can be completed at the time of issue. When this is not possible, the processor inserts sufficient empty cycles to perform the instructions precisely - explicit no-operation instructions are not required

Conclusion

[00290] Having fully described various embodiments of the invention, those skilled in the art will recognize, given the teachings herein, that numerous alternatives and equivalents exist which do not depart from the invention. It is therefore intended that the invention not be limited by the foregoing description, but only by the appended claims.